PROCESSANDO A INFORMAÇÃO

Um livro prático de programação independente de linguagem

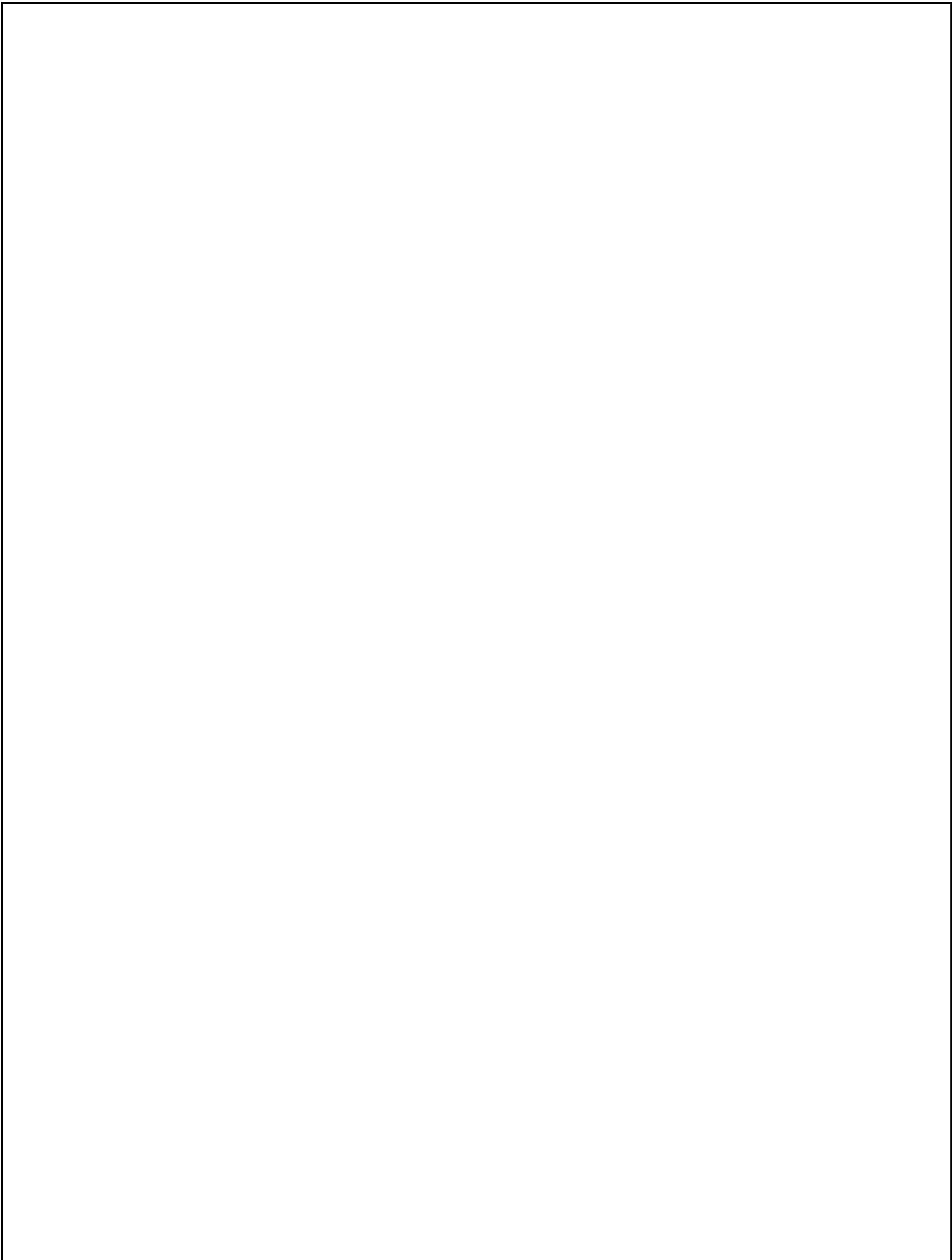


Rogério Neves

Francisco Zampirolli

Centro de Matemática, Computação e Cognição

Universidade Federal do ABC



Rogério Perino de Oliveira Neves

Francisco de Assis Zampirolli

PROCESSANDO A INFORMAÇÃO: UM LIVRO PRÁTICO DE PROGRAMAÇÃO INDEPENDENTE DE LINGUAGEM

Santo André – SP

Autor Editor: Francisco de Assis Zampirolli

2016

Diagramação e Acabamento:

Rogério Neves

CAPA E PROJETO GRÁFICO:
Rogério Neves

CATALOGAÇÃO NA FONTE

SISTEMA DE BIBLIOTECAS DA UNIVERSIDADE FEDERAL DO ABC

Responsável: xxxxxxxxxxxx – CRB: xxxx

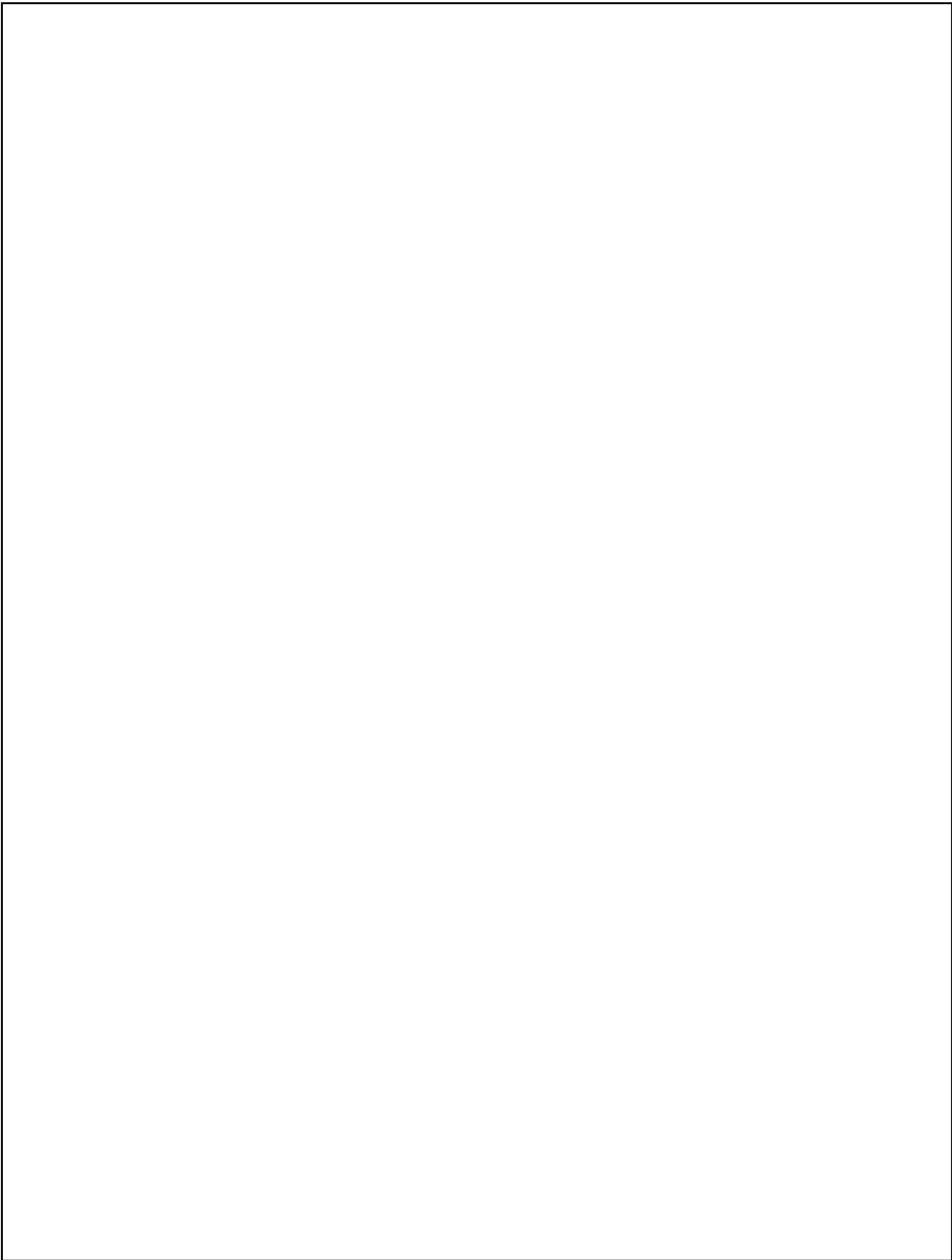
xxx.x XXXXxx	PROCESSANDO A INFORMAÇÃO: um livro prático de programação independente de linguagem / Rogério Perino de Oliveira Neves e Francisco de Assis Zampirolli – Santo André, SP: Autor Editor: Francisco de Assis Zampirolli, 2016. 136 p. il. ISBN: xxx-xx-xxxxx-xx-x 1. Programação 2. Algoritmo 3. Computação I. NEVES, Rogério Perino de Oliveira Neves II. ZAMPIROLLI, Francisco de Assis.

CONTEÚDO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE PROGRAMAS



“Programação é uma habilidade melhor adquirida pela prática e por exemplos do que por livros. ”

– Alan Turing

SOBRE ESTE LIVRO



Este livro é indicado para estudantes com alguma experiência em computação, porém, iniciantes em programação de computadores. Os conceitos são introduzidos gradualmente, organizados em capítulos e tópicos, podendo ter o aprendizado adaptado a velocidade e disponibilidade de cada leitor. Os exemplos apresentados estão em pseudocódigo, permitindo ao estudante compreender primeiro os conceitos fundamentais e escolher uma linguagem a posteriori, ou adotar uma linguagem específica dentre os exemplos apresentados de programas em linguagens convencionais.

O objetivo deste livro é proporcionar um primeiro contato entre o estudante e uma linguagem formal de programação, focando somente nos conceitos e estruturas fundamentais encontradas em qualquer linguagem de programação.

Para entender o mundo atual, onde a tecnologia nos envolve e tudo ao nosso redor contém pequenas unidades programáveis, é essencial compreender como é feita a programação, operação e comunicação destas máquinas. Computação ubíqua é um conceito cada vez mais presente em nossas vidas. Testemunhamos uma integração gradual entre a tecnologia e os objetos que nos cercam. Os computadores, cada vez mais presentes no nosso cotidiano, tendem a desaparecer em meio ao ambiente, distribuídos em utensílios domésticos, ferramentas profissionais e equipamentos de entretenimento, todos interconectados, interativos e reativos ao ambiente e ao usuário. O termo "Internet das coisas", ou simplesmente IoT, caracteriza esta transição, onde tudo está sendo conectado. Neste contexto, um indivíduo consciente do funcionamento destes dispositivos inteligentes tem uma clara vantagem, além da familiaridade e fácil adaptação a tecnologias emergentes, ele poderá, eventualmente, até fazer contribuições a elas.

Os interessados em aprimorar seu conhecimento de tecnologia podem usar este livro como referência para estudos direcionados. Também pode ser usado como complemento para o aprendizado de alguma linguagem de programação específica.

O livro ainda visa ensinar os estudantes a pensar de forma lógica e a buscar soluções para os problemas abordados utilizando algoritmos, o que o torna especialmente útil para:

- Estudantes de cursos de ciência e tecnologia
- Profissionais de diversas áreas buscando aprimoramento
- Professores, como texto de referência e sugestões de problemas práticos

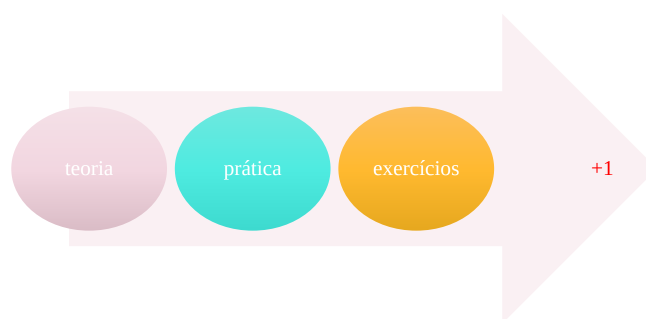
Em complemento ao conteúdo impresso, o **site de apoio** traz exemplos, programas prontos, aulas e mídias interativas, além de **referências atualizadas regularmente**,

servindo de ponto de partida para busca de material na Web. O site pode ser acessado pelo endereço http://cmcc.ufabc.edu.br/~rogerio.neves/pi/ (acesso em 2/2017). O leitor pode optar por digitar os códigos apresentados neste livro ou usar os exemplos disponíveis em formato digital, embora seja **fortemente recomendada a digitação**, que permite a familiarização do leitor com o código, encontrando possíveis erros durante sua escrita. Como se tornará familiar, em programação, caracteres no lugar errado, maiúsculos ou minúsculos trocados e até a tabulação pode resultar em erros, a maioria de difícil localização. T**estar os códigos aos poucos** (linha por linha), é mais fácil que corrigir erros do código todo. Isto também vale para os exercícios apresentados no final de cada capítulo. Uma solução de exercício pronta vale menos para o aprendizado que tentar resolvê-lo, o que permitiria a identificação de possíveis deficiências.

Como descrito anteriormente, este livro apresenta uma abordagem lógica da programação de computadores, usando fluxogramas e pseudocódigos, além de alguns exemplos em diferentes linguagens de programação para comparações. Porém, o site de apoio oferece outras versões digitais deste livro, onde além do pseudocódigo, o leitor escolhe outras linguagens de programação para a parte prática em laboratório e também para comparações. Assim, ao escolher uma linguagem de programação específica, todos os exemplos estarão disponíveis no livro digital nesta linguagem escolhida. Este recurso é útil, por exemplo, quando um professor deseja implementar em laboratório usando uma linguagem específica os exemplos e exercícios apresentados neste livro. Até o momento, oferecemos versões digitais deste livro em Java, Python e Portugol Studio.

ORGANIZAÇÃO

O livro é organizado em capítulos, cada capítulo cobrindo conceitos de programação com dificuldade progressiva, trazendo ao longo da explicação exemplos e desafios, incentivando o estudante a praticar o conteúdo assimilado. Ao final de cada capítulo, o estudante pode testar seu conhecimento temático resolvendo os exercícios propostos.



Cada capítulo pressupõe o entendimento prévio dos capítulos apresentados anteriormente, sugerindo uma ordem a ser seguida para o aprendizado, porém, podendo ser revisitados a medida do necessário para o acompanhamento dos capítulos mais avançados.

SOBRE OS AUTORES

ROGÉRIO PERINO DE OLIVEIRA NEVES

Na data desta edição, o Prof. Dr. Rogério Perino de Oliveira Neves ensina e pesquisa nas áreas de Computação Científica, Computação de Alto-desempenho (HPC), Inteligência Artificial e Robótica no Centro de Matemática, Computação e Cognição (CMCC) da Universidade Federal do ABC (UFABC). É graduado em Física Computacional pelo Instituto de Física de São Carlos da Universidade de São Paulo (IFSC-USP), mestre em Engenharia de Sistemas Eletrônicos pela Escola Politécnica da Universidade de São Paulo (POLI-USP) e doutorado em Engenharia de Sistemas pela Escola de Engenharia da Universidade Nacional de Yokohama (横浜国立大学の理工学部).

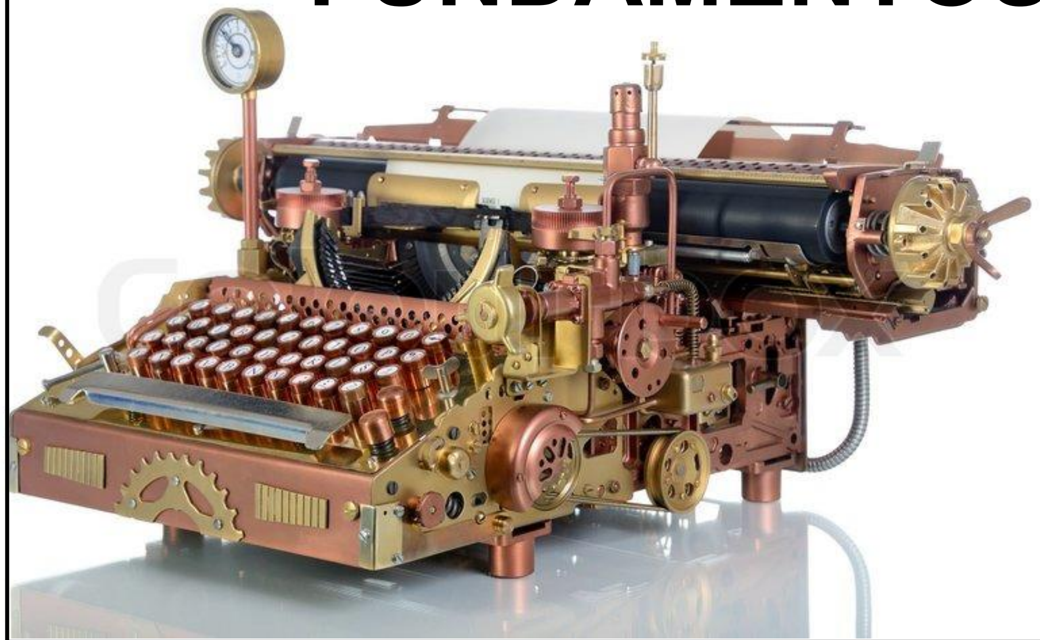


FRANCISCO DE ASSIS ZAMPIROLLI

O Prof. Dr. Francisco de Assis Zampirolli atualmente é professor no Centro de Matemática, Computação e Cognição (CMCC) da Universidade Federal do ABC (UFABC) em disciplinas de programação, engenharia de software e processamento de imagens. Possui graduação em Matemática pela Universidade Federal do Espírito Santo (UFES), mestrado em Matemática Aplicada pela Universidade de São Paulo (IME-USP) e doutorado em Engenharia Elétrica pela Universidade Estadual de Campinas (FEEC-UNICAMP). Além de lecionar disciplinas na área de Ciência da Computação, possui experiência em pesquisa e em desenvolvimento de software, atuando principalmente nos seguintes temas: processamento de imagens, visão computacional, morfologia matemática, grafos e geração automática de código e documentos.



Capítulo 1
FUNDAMENTOS



1 FUNDAMENTOS

Introdução à arquitetura de computadores

Hardware e Software

Algoritmos, fluxogramas e lógica de programação

Conceitos de linguagens de programação

Variáveis, tipos de dados e organização da memória

Operadores e precedência

Aprendendo a programar

Exercícios

Os computadores modernos, desde supercomputadores até os celulares e relógios inteligentes, são baseados na arquitetura proposta por John Von Newmann[1] em 1936. Nesta arquitetura, os computadores são divididos em elementos funcionais fundamentais. São eles: entrada de dados, processamento, saída de dados, armazenamento e comunicação de dados. Os componentes necessários para operação do computador, como entrada, saída e armazenamento, são interligados por vias de dados à Unidade Central de Processamento (CPU) ou processador, como mostrado na Figura 1.1.

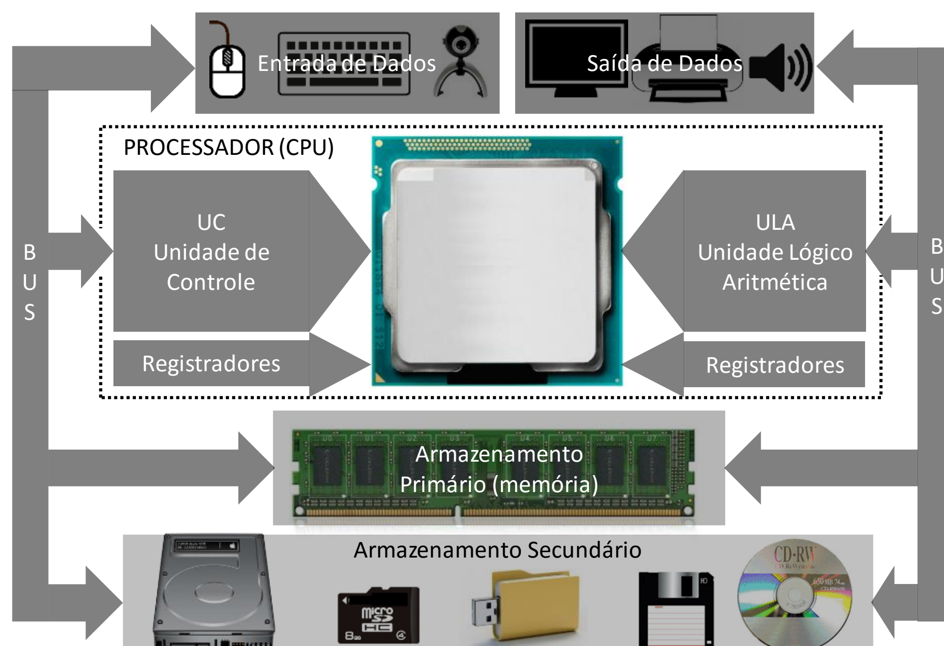


Figura 1.1. Diagrama da arquitetura de John Von Newmann.

Os dispositivos do computador estão interligados através de linhas de dados chamadas de **BUS** (ônibus, em inglês). Arquiteturas distintas podem conter várias linhas de dados paralelas, visando ampliar a velocidade de transmissão de dados entre os vários dispositivos. Algumas arquiteturas apresentam redundância em uma determinada conexão, visando aumentar a velocidade de transmissão. Alguns exemplos de linhas de dados são o padrão *QuickPath Interconnect* da Intel, o *Direct Connect* da AMD, também padrões PCI, PCI Express, USB, ISA, entre outros.

Os **dispositivos de entrada e saída** são responsáveis pela interface homem-máquina e incluem desde *joysticks* até impressoras 3D e sensores de movimento. Como exemplo, os dispositivos de entrada mais usados hoje em dia são o teclado, o *mouse*, o *joystick*, a tela

[1] Descubra mais sobre arquitetura de John Von Newmann na seção “referências” do site de apoio, em http://cmcc.ufabc.edu.br/~rogerio.neves/pi/

sensível ao toque e o microfone, enquanto os de saída mais comuns são o monitor ou tela LCD, o alto-falante e a impressora.

O **processador** ou Unidade Central de Processamento (**CPU**, em inglês) é o chip que encapsula os circuitos da **Unidade de Controle** (UC), da **Unidade Lógico-aritmética** (ULA) e **registradores**, além de circuitos de controle e comunicação internos. A UC, como se espera, controla o fluxo de dados pelas várias vias de interconexão e aciona os circuitos específicos da ULA, conforme **instruções** contidas no programa em execução. A ULA contém os circuitos responsáveis pelas operações lógicas, binárias e algébricas. Por fim, os registradores armazenam os valores de entrada e saída sendo operados pela ULA. Os registradores, implementados em circuitos eletrônicos de estado sólido (*flip-flops*), são a memória mais rápida e mais cara do computador, existindo, portando, apenas um pequeno número deles.

O acionamento dos circuitos da ULA é feito pela Unidade de Controle através de códigos específicos do programa em execução, lidos da memória sequencialmente, conhecidos como **instruções em linguagem de máquina**, sendo esta linguagem a forma mais elementar que qualquer programa assume antes de ser executado pelo processador. Estes códigos definem o circuito (ou circuitos) a serem acionados para cada **operação** requisitada, assim como quais dados devem ser operados e onde se encontram. Caso se encontrem na RAM, os dados devem ser movidos pela Unidade de Controle para os registradores antes de poderem ser utilizados pela ULA.

Em um **ciclo de instrução**, uma instrução é lida da memória, os dados necessários são **carregados das respectivas posições de memória para os registradores** e os circuitos adequados são acionados pela UC de forma síncrona, usando o relógio de pulsos ou ***clock***. Este fluxo de dados entre memória e registradores consome grande parte dos recursos do processador e tempo de processamento, criando um gargalo na comunicação de dados. Por essa razão, são empregados diversos tipos de memória para agilizar o acesso aos dados.

Os dispositivos para armazenamento (ou leitura) de dados são organizados hierarquicamente por **latência** (tempo de resposta), com custo e velocidades de acesso diferentes para cada tipo. Como mostra a pirâmide da Figura 1.2, os dispositivos mais rápidos (de menor latência) e caros estão no topo, enquanto ao descer a pirâmide encontramos os dispositivos de maior capacidade e menor custo por *byte*. Os dispositivos de entrada de dados são, geralmente, os mais lentos do ponto de vista de processamento, envolvendo perdas de milhares de ciclos ociosos na espera pela entrada do dado.

A **memória primária ou RAM** (*Random Access Memory*), que permite acesso aleatório aos dados, contém os dados que estão em uso na sessão corrente do sistema, incluindo dos aplicativos abertos e do sistema operacional. A memória RAM utiliza uma tecnologia de matriz capacitiva, com custo bem inferior por *byte*, porém latência bem mais alta que os registradores. Por esta razão, as modernas arquiteturas de computadores comumente empregam dois ou mais níveis de **memória *cache***, memórias intermediárias de maior velocidade e custo que a RAM que armazenam cópias de pequenos blocos de dados

contidos na memória principal sendo acessadas pelo processador. O *cache* pode estar dividido em vários níveis, em blocos funcionais, entre as várias unidades de processamento (*cores*) e podem ter tecnologias e tamanhos diferenciados, podendo ser tanto internos quanto externos ao chip do processador.

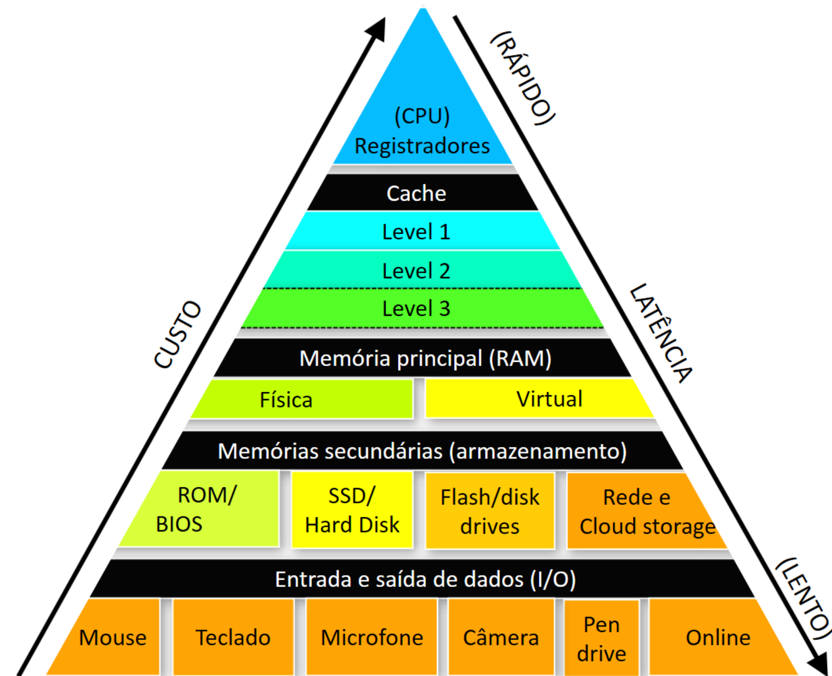


Figura 1.2. Pirâmide de dispositivos de armazenamento e entrada de dados.

Dados que não estão em uso corrente na seção são armazenados para uso futuro em dispositivos mais lentos e de menor custo por *byte*, denominados **armazenamento secundário** ou **memória secundária**. Alguns exemplos de armazenamento são discos rígidos (HDDs), *drives* de estado sólido (SSD), memórias FLASH e discos magnéticos.

A velocidade de acesso aos dados é, geralmente, o principal gargalo em programação, mas não basta investir em soluções mais rápidas e caras, é preciso entender a diferença de funcionamento e latência entre cada tipo de memória e entre cada tipo de dado nela armazenado, visando escrever programas que utilizem tais recursos de forma eficiente e obtenham um melhor desempenho. Entender como funciona o armazenamento, a comunicação e o processamento dos dados permite que o desenvolvedor construa programas mais rápidos e eficientes.

PARTES DE UM MICROCOMPUTADOR (*HARDWARE*)

Um microcomputador é composto de diversos componentes independentes, responsáveis pelo seu funcionamento, cada um cumprindo uma tarefa específica. A seguir veremos uma breve descrição dos componentes comumente encontrados em sistemas convencionais.

Gabinete é uma caixa, geralmente metálica, que salvaguarda os componentes eletrônicos mais sensíveis do microcomputador. Algumas vezes erroneamente chamado de CPU, o gabinete traz encaixes internos para a fonte de alimentação, placa mãe, *drives* (rígidos, magnéticos, óticos e de estado sólido), além de conectores para dispositivos externos.

Dispositivos de entrada e saída: são os componentes responsáveis pela interface homem-máquina. Existem diversos tipos de dispositivos para captura e geração de informação humanística, através de gráficos, som, impressões, objetos tridimensionais e movimentos. Os dispositivos de entrada e saída de dados encontram-se geralmente fora do gabinete, como mostrado na Figura 1.3.

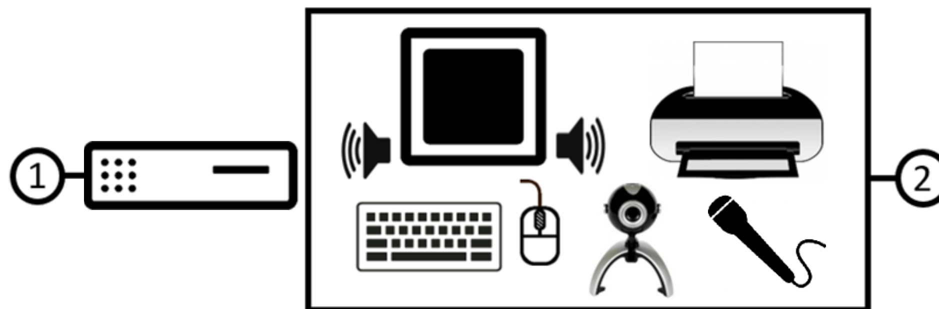


Figura 1.3. Gabinete (1) e dispositivos de entrada e saída (2).

O gabinete contém, fixados no seu interior, os dispositivos responsáveis pela alimentação e comunicação dos componentes, processamento e armazenamento dos dados. São eles:

Fonte de alimentação: um dos elementos mais importantes do computador, responsável por prover tensões operacionais constantes, requeridas pelos componentes, fornecendo corrente suficiente para alimentar todos e cada um deles, individualmente. A escolha de uma boa fonte é fundamental para a estabilidade e a durabilidade de um sistema.

Placa mãe: contém linhas de dados (BUS) e componentes de controle e operação de fluxo entre dispositivos. Neste contexto, a placa mãe serve para interligar os elementos da arquitetura de Von Newmann com outros componentes eletrônicos e eletromecânicos necessários para a operação do computador. A placa mãe contém os encaixes (*slots*) que receberão o processador (CPU), módulos de memória e placas de expansão, além de conexões para dispositivos internos e externos de diversos tipos e padrões, como IDE, SATA, USB, etc.

Memória RAM: unidade de armazenamento primário, onde serão mantidos os dados em uso corrente pelo sistema. A memória mais usada é a padrão SDRAM – Memória de Acesso Aleatório (RAM) Dinâmica e Síncrona (SD), que proporciona velocidade e confiabilidade, mas que não retém as informações quando a fonte de alimentação é desligada. Sistemas modernos usam memórias DDR (*Dual Data Rate*), que podem acessar os dados duas vezes por pulso de *clock*.

Disco Rígido, **HD** (*Hard Disk*) ou **SSD** (*Solid State Disk*): unidade de armazenamento (memória secundária) de alta velocidade, de onde serão carregados para a memória os programas e dados conforme requisitados pelo usuário e pelo sistema operacional.

Outras partes como unidades de leitura ótica e magnética, dispositivos de comunicação, visualização, expansão e refrigeração também são comumente encontrados dentro do gabinete, além de cabos e conectores de vários padrões para interliga-los.

PROGRAMAS E DADOS (*SOFTWARE*)

Steve Jobs certa vez chamou os primeiros computadores de “calculadoras glorificadas”. De fato, a única característica que distingue os computadores das máquinas existentes anteriormente é a capacidade de executar programas. Combinando um conjunto finito de instruções podemos gerar infinitos procedimentos diferentes, com resultados distintos na saída para um mesmo conjunto de dados, ou para dados diferentes. Isto foi demonstrado pelo modelo conceitual proposto pelo matemático inglês Alan Turing[2] em 1936, conhecido como a “Máquina de Turing”. Embora *software* seja um termo geralmente usado para se referir a programas, o conceito é mais abrangente, e foi originalmente usado para indicar os componentes computacionais não manipuláveis fisicamente. A parte “leve” (*soft*) da computação compreende todos os elementos do sistema manipulados unicamente através do *hardware*. São exemplos de *software*:

- **Sistemas operacionais** como Windows, Linux, iOS e Android;
- **Aplicativos** como Office, Internet Explorer, Safari e jogos;
- **Ambientes de desenvolvimento** como NetBeans, Eclipse, VisualC++ e Matlab;
- **Programas** desenvolvidos pelo usuário;
- **Dados**, como imagens, músicas, vídeos, documentos, *scripts*, etc.

Em resumo, *software* é tudo aquilo que existe no éter virtual, seja em formato eletrônico ou magnético, presente apenas no domínio da informação; é imaterial, logo, não pode ser tocado. Um software pode ser transportado com o uso de mídias, mas não pode existir sem um *hardware* que o comporte.

---

[2] Procure saber mais sobre a Máquina de Turing e as origens da computação (algumas referências no site de apoio), veja também a história do seu criador no excelente filme de 2014: “O Jogo da Imitação” (“*The Imitation Game*”).

ALGORITMOS, FLUXOGRAMAS E LÓGICA DE PROGRAMAÇÃO

ALGORITMO, definições

1. Um processo ou conjunto de regras a serem seguidas em cálculo ou outra operação de solução de problemas, especialmente por um computador.

2. Processo de resolução de um problema constituído por uma sequência ordenada e bem definida de passos que, em tempo finito, conduzem à solução do problema ou indicam que, para o mesmo, não existe soluções.

Algoritmos, em ciência da computação, são conjuntos autocontidos de operações, descrevendo passo-a-passo como realizar um processo ou resolver um problema. São geralmente listas contendo instruções (comandos), que devem ser executadas ordenadamente, de forma a solucionar um problema específico. A execução de um algoritmo ocorre após a sua codificação usando alguma linguagem (não necessariamente de programação).

Receitas são listas desse tipo, contendo **instruções** sobre como preparar um prato, onde enumeramos os passos necessários para se completar a tarefa. Através de **comandos**, uma receita indica cada ação a ser realizada, assim como o alvo da ação. Um comando é geralmente representado por um verbo, no infinitivo ou imperativo, indicando a ação a ser realizada sobre um ou mais objetos ou alvos, denominados **parâmetros**. Os parâmetros de um comando indicam os objetos (ou dados) que sofrerão a ação. As ações, no caso da receita culinária, serão, por exemplo, 'acrescente', 'misture', 'bata', 'asse', enquanto os parâmetros podem ser tanto os ingredientes utilizados quanto as medidas de quantidade, ou tempo, que acompanham cada passo.



Outro exemplo comum de algoritmo é a solução de um problema matemático. Como em um programa de computador, o estudante deve seguir o procedimento de forma ordenada para obter o resultado esperado. Por exemplo:

Solução de uma equação do segundo grau no formato $ax^2 + \mathrm{bx} + \mathrm{c}$:

1. Calcule Δ com a fórmula $\Delta = b^2 - 4ac$;
2. Se $\Delta < 0$, x não possui raízes reais;
3. Se $\Delta = 0$, x possui duas raízes reais idênticas;
4. Se $\Delta > 0$, x possui duas raízes reais e distintas;
5. Calcule x usando a equação $x = -b \pm \sqrt{\Delta}/2a$.

O algoritmo acima apresenta os passos para solução de uma equação de 2º grau, usando os **comandos** "calcule" e **condições** "se". As variáveis que aparecem nas fórmulas e os valores são os **parâmetros**. Neste caso, as fórmulas e as condições lógicas determinam o resultado a ser obtido.

FLUXOGRAMAS

Os algoritmos, além de poderem ser representados por listas de instruções, como no último exemplo, podem ser representados graficamente para facilitar seu entendimento. Os **fluxogramas** e **diagramas UML** (*Unified Modelling Language*) estão entre as representações mais usadas. Ambos, bem similares, usam formas e setas para indicar operações e seu fluxo. A Figura 1.4 ilustra uma régua com vários elementos que podem ser usados para gerar um fluxograma.

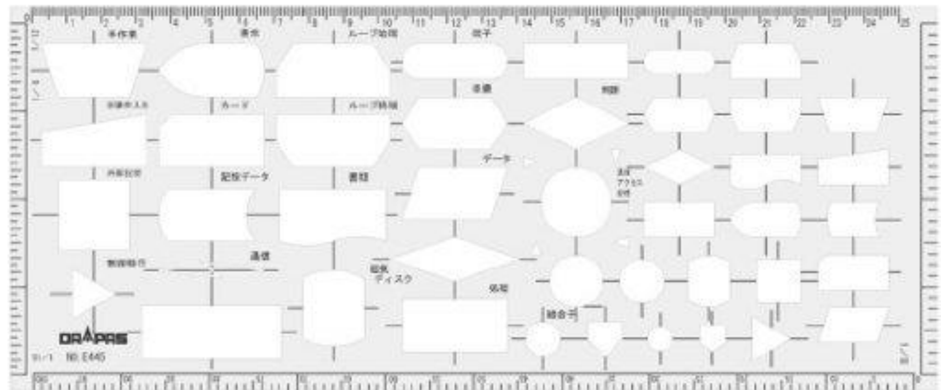


Figura 1.4. Régua de fluxograma.

A Tabela 1-1 descreve alguns desses elementos. Com eles é possível representar visualmente o fluxo de execução do programa utilizando símbolos e descritores.

Tabela 1-1: Alguns símbolos de fluxogramas.

Terminador: Indica início/fim do programa	Conector (referência): Conecta dois pontos separados na página
Processo/Operação	Referência externa: Conecta pontos em páginas distintas
Entrada e saída de dados	Impressão
Tomada de decisão: Define um desvio condicional de fluxo	Arquivo

Pode-se representar um programa conectando-se os símbolos de um fluxograma com setas, indicando o sentido do fluxo. A Figura 1.5 exemplifica o uso de fluxogramas descrevendo o processo para se assar um pão.

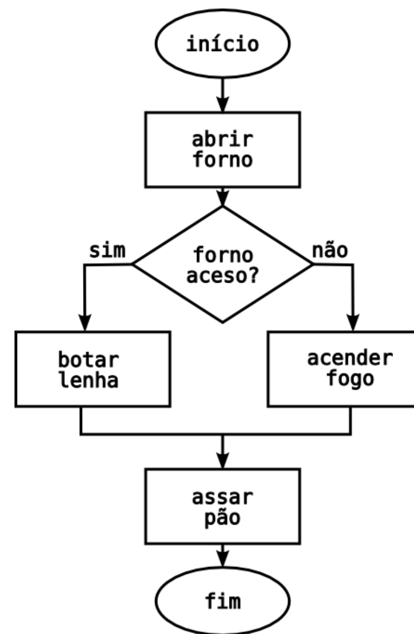


Figura 1.5. Exemplo de fluxograma.

Contudo, para se expressar um algoritmo que resolve problemas em um computador, não basta descrever os passos necessários, como em uma receita, visto que muitos problemas não podem ser resolvidos sequencialmente. Problemas complexos vão exigir o uso de ferramentas e **lógica de programação**, o que poder envolver repetições, condições de desvios e quebras de fluxo.

LÓGICA DE PROGRAMAÇÃO

A programação de computadores tem suas origens na lógica digital ou álgebra booleana[3], que por sua vez é baseada na lógica tradicional ou aristotélica.

A palavra "lógica" (λογική) se origina do grego, combinando as palavras *logos* (**razão**) e *ica* (**ciência**). Portanto, lógica propõe o estudo filosófico do raciocínio. A lógica aristotélica, com suas origens na Grécia antiga (antes de cristo), foi aprimorada por gerações de filósofos (Sócrates → Platão → Aristóteles) que aperfeiçoaram as notações e estruturas usadas para expressar o formalismo das leis de construção do raciocínio. Aristóteles, discípulo de Platão, popularizou seu conhecimento e de seus mentores através de sua detalhada obra, em 6 volumes, intitulada "*Organon*" (A Ferramenta).

> *"Nada de significante foi adicionado a lógica de Aristóteles durante dois milênios"*
> *- Immanuel Kant (Prússia)*

[3] Nome dado em homenagem ao matemático George Boole, que em 1853 introduziu o formalismo aplicado por Shannon em 1938 a circuitos digitais.

O silogismo aristotélico é o formalismo mais conhecido, onde o raciocínio é apresentado através de 3 enunciados: 2 premissas e 1 conclusão. Como exemplo:

Premissa: Alunos que estudam a literatura são bons alunos.
Premissa: Nem todos os alunos estudam a literatura.
Conclusão: Nem todos os alunos são bons alunos.

A lógica que usaremos para programar, embora não empregue a linguagem natural humana, utiliza muito do formalismo e das notações introduzidas por Aristóteles a mais de 2000 anos.

Premissa A: Todos os homens são bípedes.
Premissa B: Alguns animais são homens.
Conclusão: Alguns animais são bípedes.

$$A \wedge B \rightarrow C$$

(Se **A** e **B** então **C**)

De forma similar, a lógica de programação propõe o estudo do raciocínio necessário para se elaborar programas de computador, ou ainda, como organizar de forma lógica uma sequência de comandos a serem enviados ao processador para solucionar um problema. Se quiséssemos expressar computacionalmente o último exemplo, teríamos algo do tipo:

A = verdadeiro
B = verdadeiro
Se (A e B) então C = verdadeiro, senão C = falso

Em uma linguagem de programação, supondo que se queira estabelecer uma relação entre condições e conclusão (ou ação), teremos que utilizar a formalidade da linguagem escolhida, além de conceitos fundamentais de computação e matemática, como variáveis na memória, operadores lógicos, aritméticos e estruturação de código, a serem apresentados nos próximos tópicos. No entanto, lógica de programação é uma disciplina que exige, além do um bom conhecimento dos fundamentos, aprendizado que só pode ser obtido através da prática. A seguir serão apresentados os conceitos de linguagem, variáveis e sequência lógica de um programa e nos capítulos seguintes, cada uma das estruturas lógicas utilizadas para se expressar um programa computacional.

LINGUAGENS DE PROGRAMAÇÃO

Linguagem é um conjunto de regras usado para se definir uma forma de comunicação. São conceitos fundamentais de qualquer linguagem, não só em computação:

- **Sintaxe** – o arranjo de palavras e sua disposição em um discurso para criar frases bem formadas e relacionadas de forma lógica em uma linguagem.
- **Semântica** – o ramo da linguística preocupado com a lógica e o significado das palavras. Sub-ramos incluem a semântica formal e a semântica lexical.

Um computador não é particularmente útil a não ser que se comunique a ele, através da programação, uma sequência de comandos descrevendo em detalhes as operações que devem ser efetuadas de forma a completar uma tarefa desejada. Para programar um computador é necessário organizar os comandos sequencialmente, de forma lógica e estruturada, como em qualquer linguagem natural humana. A esta estrutura damos o nome de **sintaxe** Obedecendo a sintaxe estabelecida pela linguagem, a descrição dos comandos se dá por palavras reservadas, pertencentes ao conjunto de instruções que a definem. Este conjunto de instruções, ou **semântica** da linguagem, informa para a Unidade de Controle quais rotinas e circuitos contidos na ULA deverão ser ativados ao longo do tempo para realizar os comandos desejados.

Como exemplo, a **sintaxe** do C++ e do Java delimita funções e blocos de código usando os caracteres '{' e '}', enquanto o pascal utiliza as palavras *begin* e *end*. Trechos do código podem ser ignorados na execução do programa em linguagens de programação introduzindo-se, na frente, a sequência apropriada de caracteres indicando que se trata de um **comentário**, como, por exemplo, '//' ou '#' ou até a palavra 'REM' (*remember*) dependendo da linguagem. A **semântica** da maioria das linguagens inclui um comando '*print*' (imprimir), ou alguma variação dele, para exibir mensagens do programa na saída padrão do sistema (geralmente o monitor).

Logo, a **sintaxe** (forma) e a **semântica** (significado) definem uma **linguagem**, que permitirá expressar, usando a **lógica de programação** (que independe da linguagem escolhida), o **programa** ou sequência de **instruções** contendo os **comandos** e **expressões** algébricas, lógicas e relacionais necessárias para transformar os **dados de entrada** do programa nos **dados de saída** desejados.

As linguagens de programação são definidas usando-se uma metalinguagem para descrevê-las, geralmente empregado a forma definida por Peter Naur e aprimorada por John Backus, conhecida como forma Backus-Naur ou BNF (*Backus-Naur Form*). Cada linguagem contém suas próprias **palavras reservadas** e **lexemas** (símbolos para operadores e abstrações) usados para descrever os procedimentos computacionais realizados.

Algumas das definições das linguagens de programação, por exemplo, formalizam a estrutura das funções, uso de variáveis, expressões algébricas, expressões booleanas, abstrações para manipulação de estruturas de dados e controles de fluxo. Estes conceitos

serão apresentados gradualmente ao longo do livro para várias linguagens, para comparação. A complexidade da linguagem vai depender da similaridade entre estas estruturas e abstrações com as comumente usadas por nós (por exemplo, a matemática).

NÍVEIS DE LINGUAGEM

Nível de linguagem diz respeito à proximidade que a mesma se encontra da linguagem natural humana. É considerada de alto nível uma linguagem com forte abstração do funcionamento do computador, portando, de fácil compreensão para um ser humano. Desta forma, quanto mais fácil a compreensão do código por um ser humano que fale o idioma usado na linguagem, de mais alto-nível ela é considerada (embora esta definição possa ser questionada por futuras máquinas pensantes). A lista é encabeçada por linguagens visuais com sintaxe e semântica similares ao idioma Inglês e termina nos códigos em binário usados para acionamento dos circuitos da ULA (máquina), como mostra a Figura 1.6.

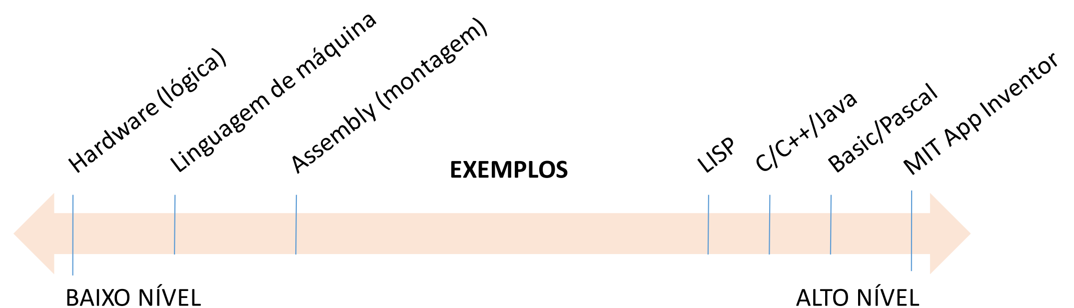


Figura 1.6. Níveis de linguagem de programação.

De acordo com a definição, são linguagens de baixo nível aquelas próximas ao modo operacional do hardware, sem qualquer tipo de abstração, sendo o nível mais baixo a programação direta em hardware através de portas lógicas. Em seguida vem a linguagem de máquina, que é uma representação numérica (binária) dos códigos de acionamento de circuitos internos específicos do computador, assim como valores a serem atribuídos a entradas e saídas destes circuitos.

A dificuldade de se trabalhar com código em máquina através da programação por *bits* fez com que os engenheiros da época desenvolvessem programas para montar código binário com maior facilidade[4], nascem as linguagens de montagem. A linguagem de montagem, conhecida como *Assembly* ou *Assembler* é uma representação dos códigos da linguagem de máquina através de palavras chaves e com parâmetros representados por números, geralmente em hexadecimal.

---

[4] O site de apoio, http://cmcc.ufabc.edu.br/~rogerio.neves/pi/ traz um vídeo mostrando a programação do EDSAC 1951, um dos primeiros computadores programáveis.

A representação hexadecimal é vantajosa em programação por permitir representar conjuntos de quatro *bits* ou um “*nibble*” (meio-*byte*) através de um único caractere. Por exemplo, um *byte* é representado por dois caracteres em hexadecimal (base 16) e pode assumir valores de 00_{16} a FF_{16} ($0000\ 0000_2$ a $1111\ 1111_2$ ou 0_{10} a 255_{10}).

Independentemente do nível de linguagem em que foi escrito, para ser executado, todo o programa deve ser traduzido para a linguagem da máquina. Esta interpretação pode ser feita em modo de execução, onde o código é interpretado à medida que é executado, ou previamente, gerando um código executável (EXE) final, na linguagem de máquina nativa do sistema ao qual se destina. Este processo de gerar o código de máquina é conhecido como **compilação**. Compilar o código significa gerar um código executável em linguagem de máquina, onde cada família de processadores tem seu padrão próprio.

LINGUAGEM COMPILADA X INTERPRETADA

A **compilação** é o nome dado ao processo de traduzir um código escrito em uma linguagem de mais alto-nível para código de máquina, que poderá ser executado na arquitetura apropriada para o qual foi compilado. O resultado de compilar um programa é um arquivo binário contendo o programa na linguagem nativa do processador (máquina). Exemplos de linguagens compiladas são C, C++, Pascal e Fortran.

Em **linguagens interpretadas**, os comandos do programa são transformados em código nativo durante a sua execução, geralmente por um outro programa, necessário para seu funcionamento. Exemplos comuns são JavaScript, Matlab, Portugol, Python e PHP.

Natural assumir que um código compilado para uma determinada arquitetura, embora mais eficiente, só poderá rodar em computadores compatíveis com esta arquitetura, enquanto que qualquer computador que dê suporte a uma linguagem interpretada poderá executar qualquer programa escrito nela. O resultado é que programas compilados tem baixa **portabilidade**, ou seja, não cobrem um grande número de **plataformas** de execução, enquanto um programa interpretado, geralmente, tem maior portabilidade.

A exceção é a linguagem **Java**, que compila um código binário em *bytes* (***bytecode***), para ser executado não em uma arquitetura real específica, mas sim pela **Máquina Virtual Java** (JVM), que é um **programa nativo** (isto é, é necessário um JVM compilado para cada arquitetura) responsável por transformar *bytecode* na linguagem de máquina nativa do processador em tempo de execução (***runtime***). A existência de Máquinas Virtuais Java para a maioria das plataformas aumentou em muito a portabilidade dos programas. A **portabilidade** foi umas das principais motivações para a criação da linguagem Java.

ESTRUTURAS DE CÓDIGO

Independente da linguagem escolhida, as estruturas fundamentais de código que estarão presentes em todas elas são:

- **Código sequencial**: os comandos são executados na ordem em que aparecem;
- **Módulos de código** (funções, métodos ou classes): conjuntos de comandos agrupados em sub-rotinas ou subprogramas;
- **Desvios condicionais**: dada uma condição, existem dois caminhos possíveis (duas linhas) de execução;
- **Estruturas de repetição ou laços**: conjuntos de comandos que se repetem enquanto uma dada condição for satisfeita.

Cada um destes temas será tratado individualmente em um capítulo distinto deste livro. Alguns tópicos avançados serão apresentados no último capítulo, como sugestão para estudos subsequentes, como Programação Orientada a Objetos e Engenharia de Software, que mostram como organizar seu código em objetos e como construir sistemas computacionais, especialmente quando trabalhando em equipe.

DEPURAÇÃO DE CÓDIGO (DEBUG)

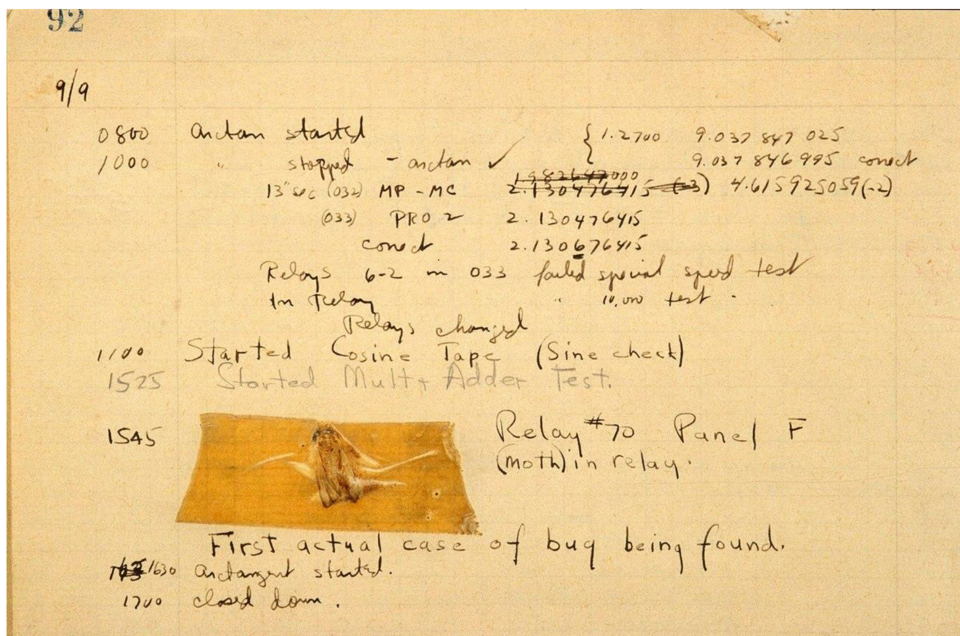
É referido como depuração ou "debugação" o árduo processo de busca e correção dos erros de programação no código escrito em linguagens de programação. Ferramentas de depuração são de grande valia para auxiliar na busca e correção destes erros.

Alguns dos erros comuns encontrados em código, do menos grave ao mais sério, são:

- **Erro de semântica** – palavras reservadas ou operações escritas de forma errada;
- **Erro de sintaxe** – envolve o uso inapropriado do formato dos comandos;
- **Erro de organização** – blocos de código, aspas ou uso de parênteses inconsistentes – ocorre quando se abre um bloco de código ou de operadores (com chaves, colchetes ou parênteses) ou um campo de texto (com aspas) sem fechar ou se fecha sem abrir;
- **Erro de lógica** – também conhecido como *Cerberus* (o cão de guarda do inferno), ocorre quando os comandos estão sintaticamente corretos, na semântica correta, e escritos consistentemente, porém em ordem, forma ou disposição que produz um resultado diferente do desejado. Resultado este que, frequentemente, causa travamento da execução do programa ou até mesmo do computador.

Faz parte da crença popular que a primeira referência ao termo *bug* usada como erro de programa ocorreu quando Grace Hopper encontrou uma mariposa presa ao relê do computador Mark II da Universidade de Harvard, impedindo o programa de rodar corretamente. No entanto, a expressão era usada muito antes disso, como por soldados na II Guerra Mundial para descrever problemas com o equipamento. Thomas Edison, em 1878 numa carta para Theodore Puskas, usou o termo "*bug*" para descrever defeitos em suas invenções:

> *"Ocorre meio que o mesmo em todas as minhas invenções. O primeiro passo, uma intuição, vem como uma explosão, então surgem as dificuldades — a coisa degringola e então aqueles insetos [bugs] — como chamamos pequenas falhas e dificuldades — aparecem e meses de intensa observação, estudo e trabalho são requeridos antes do sucesso (ou falha) comercial ser seguramente atingido"*


92

9/9

0800 Antan started

1000 " stopped - antan ✓ 9.037 847 025 / 9.037 846 995 correct

(033) MP - MC 4.615925059(-2)

(033) PRO 2 2.130476415

correct 2.130676415

Relays 6-2 in 033 failed special speed test

In relay " 10.000 test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F (moth) in relay.

First actual case of bug being found.

1630 Antangent started.

1700 closed down.

Figura 1.7. Página do diário de Grace Hopper com o "*bug*" encontrado no Mark II.

Ferramentas de desenvolvimento geralmente oferecem alguma forma de depurar o código, indicando também palavras com sintaxe incorreta, as linhas contendo erros e o tipo de erro ocorrido durante a execução.

AMBIENTES DE DESENVOLVIMENTO INTEGRADOS

Ambientes de desenvolvimento integrado ou IDE (*Integrated Developing Environment*), como são conhecidas, oferecem uma combinação de ferramentas úteis para escrita, execução e depuração do código de programas. Geralmente trazem um editor de texto inteligente que realiza a marcação de palavras reservadas, variáveis, indicando erros e efetuando a tabulação do código automaticamente, alguns até auto-completam comandos. Comumente apresentam algum tipo de integração com um compilador e ajudam na depuração, retornando informações úteis durante e após a execução.

Algumas ferramentas de desenvolvimento populares do tipo IDE são:

- **Eclipse** (C, C++, Java, PHP, Android, etc);
- **NetBeans** (C, C++, Java, Ruby, HTML5, PHP, etc.);
- **Xcode** (IDE de desenvolvimento para dispositivos Apple);
- **Visual Studio** (C, C++, C#, Visual Basic, entre outros, voltado para o desenvolvimento de programas na plataforma Windows).

Tabela 1-2: Os 10 editores de código mais populares[5].

%	Codeanyware.com
28%	Notepad++
23%	Sublime Text
18%	Eclipse
8%	Netbeans
8%	IntelliJ
4%	Vim
4%	Visual Studio
4%	Php Storm
3%	Atom
2%	Emac
1%	Zend

%	Stackoverflow.com
36%	Notepad++
36%	Visual Studio
31%	Sublime Text
26%	Vim
23%	Eclipse
17%	IntelliJ
13%	Android Studio
13%	Other
12%	Atom
10%	Xcode
8%	Netbeans

Os programas apresentados neste livro foram desenvolvidos com a ferramenta educacional **Portugol** Studio para código em português e a IDE **NetBeans** para desenvolvimento em Java e C, entre outras.

[5] Levantamentos realizados pelos sites codeanywhere.com e stackoverflow.com, respectivamente (https://blog.codeanywhere.com/most-popular-ides-code-editors e http://stackoverflow.com/research/developer-survey-2016).

VARIÁVEIS



Uma variável em programação de computadores é um indicador ou 'apelido' (*alias*) atribuído à um endereço de memória que contém um dado guardado. Os elementos na memória, representados por variáveis, podem conter apenas um, ou múltiplos valores e são codificados na memória de acordo com o tipo específico do dado, com formato e número de bits apropriado para mantê-lo.

Como analogia, podemos imaginar que endereços de memória contendo variáveis são como gavetas onde armazenamos notas. Cada gaveta armazena um único dado, de um tipo específico, embora algumas variáveis possam ser formadas por vários valores numéricos e alfanuméricos, como no caso de vetores, matrizes, sequencias de caracteres e listas estruturadas.

Para o programador, o nome da variável é meramente um lembrete sobre a natureza do dado contido no endereço de memória referido, como etiquetas que colamos em um compartimento para nos lembrar do seu conteúdo. No entanto, quando compilado (ou interpretado) o código, estes nomes serão substituídos pelos endereços de memória, geralmente representados em número hexadecimal, de cada uma das variáveis.

USO DE VARIÁVEIS

Em qualquer linguagem, um valor pode ser atribuído a uma variável através do **Operador de Atribuição**. No Java e C/C++ entre outras linguagens, o operador de atribuição é o sinal de igual **'='**. A maneira de se utilizar o Operador de Atribuição é indicando, primeiramente, o nome da variável que receberá um valor, seguindo do sinal '=' e o valor a ser gravado na posição da memória alocada automaticamente para a variável.

```
IDADE   = 21;
ANO_ATUAL = 2016;
```

A leitura correta para o código representado acima é "**a variável IDADE recebe o valor 21**" e "**a variável ANO_ATUAL recebe o valor 2016**". Uma vez atribuído valor a uma variável, este valor pode ser utilizado em futuras operações através do nome da variável.

```
ANO_DE_NASCIMENTO  = ANO_ATUAL - IDADE;
```

A tentativa de se utilizar uma variável cujo valor ainda não foi atribuído resultará erro ou no uso de em um valor arbitrário (valor preexistente na posição de memória usada pela variável), dependendo da linguagem.

Um grande número de linguagens exige que uma variável seja previamente **definida** e **inicializada**, especificando, desta forma, o **tipo da variável** e o **valor inicial**.

Código 1.1. Exemplo de inicialização de variáveis em Java/C++.

```
// variável do tipo número inteiro com valor inicial 16
int idade = 16;

// variável real com ponto flutuante de simples precisão
float peso = 80.25f;  // <-- note o caractere f

// variável real com ponto flutuante de dupla precisão
double pi = 3.1415926535897932;

// cadeia de caracteres
String nome = "Luís Paulo";

// Tipo booleano (único bit, verdadeiro ou falso)
boolean flag = true;

// Apenas definição dos tipos das variáveis
int ano_de_nascimento, ano_atual;
// OBS: As variáveis ainda precisam ser inicializadas

// Inicialização das variáveis acima
ano_atual = 2016;
ano_de_nascimento = ano_atual - idade;
// as variáveis só podem ser definidas uma vez
// após a definição, basta mudar seu valor
ano_atual = 2017;
```

Existem dois tipos de variáveis para números reais em Java e C/C++: `float` e `double`. Ambos são representados na memória usando a norma **IEEE 754**, onde o ponto é flutuante (*floating point*) mas o número de *bits* varia, sendo, em geral, 32 bits para `float` (simples precisão) e 64 bits para `double` (dupla precisão). Em geral, simples precisão é adequada para a maioria dos cálculos, embora algumas aplicações recomendem o uso de `double`, devido a propagação de erros de arredondamento. Um número do tipo `float` deve ser seguido do caractere '**f**', caso contrário, será assumido como de dupla precisão. Em C, a declaração `string` é com 's' minúsculo, indicando uso da biblioteca homônima, que deve ser incluída no início do programa, usando o comando de incluir biblioteca `#include <string>`. Alternativamente se pode usar cadeias de caracteres do tipo `char`.

NOMENCLATURA

Os nomes das variáveis podem conter letras, números e alguns caracteres especiais, desde que não sejam empregados pelo lexema da linguagem (símbolos dos operadores matemáticos, lógicos e relacionais, por exemplo). Em geral, o nome da variável deve começar por uma letra ou, dependendo da linguagem, algum caractere especial. Como regra geral os nomes de variáveis não devem começar com números ou conter espaços. Caracteres especiais não devem ser usados no nome da variável a não ser como notação específica. Algumas linguagens também não aceitam caracteres contendo acentuação. Caso o nome escolhido seja inválido, a execução do código produzirá algum tipo de erro.

Lembrando que **o nome da variável serve somente para o desenvolvedor**, como forma de **apontar os endereços de memória** a serem operados, é recomendável sempre utilizar nomes **significativos** e úteis para o entendimento do programa. Uma variável deve indicar seu propósito ao manipular a memória, apontando os endereços (gavetas) que contém determinados valores, facilitando o entendimento e manutenção do código.

Alguns exemplos de nomes de variáveis que podem ou não ser usados em Java e C são:

<table>
<tr><th colspan="2">Código 1.2. Exemplo de variáveis em Java/C/C++[6]</th></tr>
<tr><th>Válido</th><th>Inválido</th></tr>
<tr><td>int idade4 = 50;
String Nome_Completo;
float _resto;
boolean resta_um;
double V4R14V3L_V4L1D4;
double R$;</td><td>int 45dias;
String Nome Completo;
float %resto;
boolean resta-um;
double V@R!@V&L+|NV@L|D4;
double N#;</td></tr>
</table>

Você saberia dizer por que os nomes à direita são inválidos?

CARACTERES RESERVADOS E OPERADORES

As linguagens reservam alguns caracteres para uso em operadores ou indicadores de tipo. Estes caracteres não podem ser usados com outro propósito, a não ser em cadeias de caracteres (*strings*), delimitadas com “aspas”. São exemplos de caracteres reservados por operadores em Java, C e C++:

- **Operadores aritméticos**:
  - + (soma com)
  - - (subtração por)
  - * (multiplicação por)
  - / (divisão por)
  - % (resto da divisão por)
  - ++ (incremento)
  - -- (decremento)
- **Operadores relacionais**:
  - == (é igual a)
  - != (é diferente de)
  - > (é maior que)
  - < (é menor que)

---

[6] Em C++ a declaração de `string` é com ‘s’ minúsculo, indicando uso da biblioteca homônima, que deve ser incluída no código com `#include <string>`.

    - >= (é maior ou igual a)
    - <= (é menor ou igual a)
- **Operadores lógicos binários**:
    - && (e), & (e bit-a-bit)
    - || (ou), | (ou bit a bit)
    - ! (não lógico ou complemento)
    - ~ (complemento bit-a-bit)
    - ^ (ou exclusivo 'XOR' bit-a-bit)
    - << N (*shift-left*, adiciona N zeros à direita do número binário)
    - >> N (*shift-right*, elimina N dígitos a direita do número binário)
- **Operadores de atribuição**:
    - = (recebe o valor de)
    - += (é somado ao valor de)
    - -= (é subtraído do valor de)
    - *= (é multiplicado pelo valor de)
    - /= (é dividido pelo valor de)
    - %= (recebe o resto da divisão por)
    - Similarmente: >>=, <<=, &=, |=, ^=
- **Operadores mistos**:
    - ? : (condicional)
- **Todos as formas de parênteses**: ([{ }])
- **Pontuação:** " : ; . , ", etc.

PRECEDÊNCIA DE OPERADORES

Quando utilizados em uma expressão, os operadores apresentados acima são executados em ordem de precedência, de forma semelhante como fazemos em equações matemáticas, com somas, subtrações, multiplicações, divisões, exponenciais, etc. Sabemos que o expoente é calculado antes das multiplicações e divisões e estas duas últimas antes das somas e subtrações. Quando com mesma precedência, as operações representadas pelos operadores são efetuadas na ordem em que aparecem.

A tabela a seguir traz a precedência de operadores para a maioria das linguagens, como Java, JavaScript, Python, C, C++ e C#, entre outras, assim como o sentido de associação na execução de mesma precedência. Algumas operações podem variar de sintaxe ou forma de associação em diferentes linguagens.

<table>
<tr><th colspan="3">Tabela 1-3. Precedência de Operadores.</th></tr>
<tr><th>Categoria</th><th>Operador</th><th>Associatividade</th></tr>
<tr><td>Pós-fixado</td><td>() . (parênteses, operador ponto)</td><td>→ Esquerda para a direita</td></tr>
<tr><td>Índice</td><td>[]</td><td>→ Esquerda para a direita</td></tr>
<tr><td>Unário</td><td>++ - - ! ~ (incremento, decremento)</td><td>← Direita para a esquerda</td></tr>
<tr><td>Multiplicativo</td><td>* / % (vezes, dividido, resto)</td><td>→ Esquerda para a direita</td></tr>
<tr><td>Aditivo</td><td>+ - (soma, subtração)</td><td>→ Esquerda para a direita</td></tr>
<tr><td>Shift binário</td><td>>> >>> <<</td><td>→ Esquerda para a direita</td></tr>
<tr><td>Relacional</td><td>> >= < <=</td><td>→ Esquerda para a direita</td></tr>
<tr><td>Igualdade</td><td>== != (é igual a, é diferente de)</td><td>→ Esquerda para a direita</td></tr>
<tr><td>E (AND) bit-a-bit</td><td>&</td><td>→ Esquerda para a direita</td></tr>
<tr><td>XOR bit-a-bit</td><td>^</td><td>→ Esquerda para a direita</td></tr>
<tr><td>Ou (OR) bit-a-bit</td><td>|</td><td>→ Esquerda para a direita</td></tr>
<tr><td>E lógico</td><td>&&</td><td>→ Esquerda para a direita</td></tr>
<tr><td>Ou lógico</td><td>||</td><td>→ Esquerda para a direita</td></tr>
<tr><td>Condicional</td><td>?:</td><td>← Direita para a esquerda</td></tr>
<tr><td>Atribuição</td><td>= += -= *= /= %= >>= <<= &= ^= |=</td><td>← Direita para a esquerda</td></tr>
</table>

TIPOS DE DADOS

Quando definimos uma variável, na maioria das linguagens, devemos indicar o tipo de dado a ser armazenado no espaço de memória reservado. Isto vai instruir o processador sobre quanto espaço deverá ser alocado na memória para ele, como será apresentado, codificado e operado. Cada tipo de dado usa um número distinto de *bits*, logo as "gavetas" devem ter "alturas" diferenciadas, assim como a ULA tem circuitos eletrônicos diferentes dedicados a processar cada tipo de dado. A tabela a seguir apresenta os tipos de dados mais comuns na maioria das linguagens de programação, a quantidade de memória ocupada por cada um e seus limites, que podem variar entre linguagens:

Tabela 1-4. Tipos de variáveis em Java/C/C++.				
`tipo`	descrição	*bits*	mínimo	máximo
Tipos inteiros com sinal				
`byte`	inteiro de 1 byte	8	-128	127
`short`	inteiro curto	16	-2^{15}	$-2^{15}-1$
`int`	inteiro	32	-2^{31}	$-2^{31}-1$
`long`	inteiro longo	64	-2^{63}	$2^{63}-1$
Tipos reais com ponto flutuante no padrão IEEE 754				
`float`	precisão simples	32	2^{-149}	$(2-2^{-23})2^{127}$
`double`	precisão dupla	64	2^{-1074}	$(2-2^{-52})2^{1023}$
Tipos lógicos				
`boolean`	valor booleano	1	*false*	*true*
Tipos alfanuméricos				
`char`	caractere unicode	16	0	$2^{16}-1$
Classe cadeia de caracteres				
`string`	sequência de n chars	16*n	-	-
Outras				
`void`	variável vazia	0	-	-

Em outras linguagens, geralmente o tipo de dado é definido pela atribuição de um valor inicial, sendo que a variável será em diante tratada como do tipo especificado pelo primeiro valor. A atribuição posterior de um valor de outro tipo envolve a alocação de outro espaço de memória. Em algumas linguagens antigas, os valores são mantidos como do tipo original, possivelmente causando perdas de precisão ou conversões de real para inteiro. No entanto, sempre é possível converter valores entre diferentes tipos de variáveis, como é exemplificado na Tabela 1-6.

Tabela 1-5. Exemplos de conversão de valores entre tipos de variáveis.

Conversão	**Método**	**Exemplo**
	C/C++/Java	
float → int	Type cast	`int i = (int) varFloat;`
double → float	Type cast	`float f = (float) varDouble;`
float, int → double	Direto	`double d = i; d = f;`
Número → Sting	(Java) direto (C) stdlib.h	`String str = "" + f;` `str = snprintf(num);`
String → int	(Java) parse (C) stdlib.h	`i = Integer.parseInt(str);` `i = atoi(str);`
String → float	(Java) parse (C) stdlib.h	`f = Float.parseFloat(str);` `f = atof(str);`
String →double	Java) parse (C) stdlib.h	`d = Double.parseDouble(str);` `d = strtod(str, NULL);`
	JavaScript	
String →int	Parse	`var i = parseInt(str);`
String →Float	Parse	`var f = parseFloat(str);`
Número → String	toString()	`Var str = toString(f);`

TESTES DE MESA

Teste de mesa é o nome dado a simulação manual da execução de um programa, acompanhando o estado das variáveis e a mudança temporal de seus valores, quando feito no papel ou mesmo mentalmente. Geralmente anota-se o nome das variáveis, a medida em que aparecem, e seus respectivos valores. Quando as variáveis são modificadas, os novos valores vão substituindo os anteriores, que são atualizados (riscados) na tabela para cada nova instrução que as modifica. Os valores finais das variáveis, ao termino do programa, são os últimos valores assumidos por cada uma delas. Segue um exemplo de teste de mesa:

```
c = 1; f = 22;
ano = 1994;
idd = 0;
ano = ano + idd;
idd *= f;
c++;
ano = ano + f;
idd += f;
f = f - 4;
c++;
```

c	f	ano	idd
~~1~~	~~22~~	~~1994~~	~~0~~
~~2~~	18	~~1994~~	~~0~~
3		2016	22

Figura 1.8. Exemplo de teste de mesa.

APRENDENDO A PROGRAMAR

Uma sugestão prática para experimentar algumas linguagens de programação é começar escrevendo um programa bem simples. Considerando que este será o primeiro passo de muitos leitores no universo da programação, deixamos algumas opções de linguagens para escolha. Os primeiros passos são os mais importantes na escolha do caminho a seguir, então, começaremos com o mais básico.

ALÔ, MUNDO

É tradição entre programadores, ao se iniciar no universo da programação em qualquer linguagem, começar com esta mensagem de boas-vindas para se familiarizar com o modo saída de dados na tela. Alô mundo é o ponto de partida para a maioria dos programadores novatos. O exercício consiste em escrever um programa completo que exiba na tela a mensagem “Alô, mundo!”. Esta atividade pode ser feita em qualquer linguagem, porém, dê preferência àquela na qual pretende desenvolver seus estudos.

Código 1.3. Alô mundo em pseudocódigo.

```
escreva(“Alô, mundo!”)
```

Qualquer um dos dois exemplos a seguir pode ser testado gravando-se o código em um arquivo com extensão ‘`.html`’ e abrindo-o em qualquer navegador.

Código 1.4. Alô mundo em HTML.

```
<!DOCTYPE html>
<html><body>
<p>Alo, mundo!</p>
</body></html>
```

Código 1.5. Alô mundo em JavaScript.

```
<!DOCTYPE html>
<html><body>
<script type="text/javascript">
  document.write ("Alo, mundo!");
</script>
</body></html>
```

Note que existe uma estrutura padrão comum a cada linguagem (código em cinza). Esta estrutura (ou alguma variação dela) se repete em todos os programas desenvolvidos na linguagem. Alguns códigos apresentados neste livro vão omitir esta estrutura, embora ela deva estar presente para seu funcionamento. Observe esta estrutura no programa em Java:

Código 1.6. Alô mundo em Java.

```
public class Alomundo {
    public static void main(String[] args) {
        System.out.println("Alo, mundo!");
    }
}
```

Para experimentar o código em Java é necessário instalar o Java JDK (*Java Development Kit*) em qualquer computador, caso este ainda não tenha sido instalado. Em seguida, siga o procedimento:

1. Salve as linhas de código em um arquivo texto com o **mesmo nome da classe** e a extensão '`.java`'. No caso, o arquivo deverá chamar-se "`Alomundo.java`";
2. No **console** (`cmd.exe` no Windows) ou **shell** (`ssh` no Linux), compile o código fonte com o comando "`javac Alomundo.java` ";
3. Execute o programa com o comando "`java Alomundo`".

Os exemplos a seguir requerem compiladores ou interpretadores para as linguagens apresentadas. O procedimento muda de acordo com a ferramenta usada.

Código 1.7. Alô mundo em C.

```
#include<stdio.h>

main(){
    printf("Alo, mundo!");
}
```

Código 1.8. Alô mundo em Python.

```
print("Alo, mundo!")
```

O Python oferece interpretadores online, onde basta digitar as instruções e observar o resultado. Experimente em www.python.org/shell/, www.pythonanywhere.com e repl.it/languages/python (IDE online, múltiplas linguagens).

Código 1.9. Alô mundo em Pascal.

```
program Alomundo;
begin
  writeln ('Alo, mundo!')
end.
```

Código 1.10. Alô mundo em Fortram77.

```
program alomundo77
print *,"Alo, mundo!"
end program alomundo77
```

Código 1.11. Alô mundo em Fortram90.

```
program alomundo90
    write(*,*) "Alo, mundo!"
end program alomundo90
```

PROGRAMAÇÃO SEQUENCIAL

Programação incorpora conceitos de matemática e de lógica, entre eles, variáveis e expressões algébricas. Como na matemática, a expressão a seguir produzirá C = 10.

Código 1.12. Pseudocódigo: Operação com variáveis[7]

```
A = 2
B = 3
C = (A + B) * 2
```

Note que, como na matemática, o resultado seria outro sem os parênteses (C = 8). Podemos incrementar o programa acima incorporando interação com o usuário, através de instruções para entrada e saída de dados:

Código 1.13. Pseudocódigo: Entrada e saída de dados

```
A = leia("Entre com o valor de A: ")
B = 3
C = 2 * (A + B)
escreva(C);
```

O programa agora apresentará a mensagem inicial "Entre com o valor de A" e **aguardará** até que o usuário entre com um valor (com número + ENTER) que será recebido pela variável, antes de executar os comandos seguintes. No final, após calcular o valor de `C`, o mesmo será exibido para o usuário na saída padrão do sistema, abaixo da mensagem inicial:

```
Entre com o valor de A: 22
50
```

Considerando o exemplo anterior, o que faz o código a seguir?

Código 1.14. Pseudocódigo: média de 3 notas

```
nota1 = leia("Digite a nota 1:");
nota2 = leia("Digite a nota 2:");
nota3 = leia("Digite a nota 3:");
media = (nota1 + nota2 + nota3)/3;
escreva("Média=" + media);
```

O código é de simples compreensão, por estar em uma linguagem conhecida (português), usar uma familiar estrutura sequencial e empregar expressões matemáticas às quais estamos acostumados. Após pedir três números para o usuário (notas 1, 2 e 3), o programa calculará a média das três variáveis e imprimirá o resultado na tela.

O programa anterior pode ser facilmente traduzido para qualquer linguagem de programação, bastando adaptar sua sintaxe e semântica para que seja compreendida

[7] Notações para pseudocódigo podem variar de acordo com o algoritmo, a linguagem, estilos de escrita e preferência, cabendo ao programador escolher a que representa seu raciocínio lógico de forma mais adequada.

pela nova linguagem. Por exemplo, se quiséssemos rodar o programa no Portugol Studio[8] precisaríamos incluir as formalidades da sintaxe e empregar a semântica apropriada, adotada pelos criadores da linguagem.

Código 1.15. Portugol Studio: Média de 3 notas

```
programa {
inicio() {
   real nota1, nota2, nota3
   escreva("Digite a 1a nota:")
   leia(nota1)
   escreva("Digite a 2a nota:")
   leia(nota2)
   escreva("Digite a 3a nota:")
   leia(nota3)
   media = (nota1 + nota2 + nota3)/3
   escreva("Média=", media)
}}
```

A linguagem JavaScript é usada por todos os navegadores para incluir processos dinâmicos em páginas web. Geralmente o código do programa está contido nas próprias páginas, logo a linguagem inclui alguns elementos de sintaxe (*tags*) do HTML (que não é uma linguagem de programação):

Código 1.16. JavaScript: Média de 3 notas[9]

```
<script type="text/javascript">
var nota1=parseFloat(prompt("Digite a 1a nota:"));
var nota2=parseFloat(prompt("Digite a 2a nota:"));
var nota3=parseFloat(prompt("Digite a 3a nota:"));
var media = (nota1 + nota2 + nota3)/3.0;
document.write ("Média=" + media);
</script>
```

Código 1.17. Pascal: Média de 3 notas

```
var nota1, nota2, nota3, media: real;
begin
    writeln('Digite a 1a nota:');
    readln(nota1);
    writeln;
    writeln('Digite a 2a. nota:');
    readln(nota2);
    writeln;
    writeln('Informe a 3a. nota:');
    readln(nota3);
    writeln;
    media := (nota1 + nota2 + nota3) / 3;
    write('Media final = ', media:0:2);
end;
```

[8] O Portugol Studio está disponível em https://sourceforge.net/projects/portugolstudio.
[9] JavaScript podem ser experimentado usando IDEs online (repl.it/languages/javascript) ou diretamente do seu navegador pelo **Console**. Por exemplo, para abrir o console no Google Chrome digite “Control-Shift-J” (“Command-Option-J” no Mac), ou no Microsoft Edge digite “F12”, ou consulte a página de ajuda do seu navegador sobre console.

Como visto, independente da linguagem escolhida, o código poderá ser traduzido entre linguagens, bastando reduzi-lo a uma estrutura básica contendo apenas a lógica de programação, comum a todas as linguagens de programação (usando fluxogramas ou **pseudocódigo)**, então empregar a semântica e sintaxe da linguagem alvo. Por exemplo, para traduzir pseudocódigo em português para Java:

Tabela 1-6. Exemplos de equivalência entre pseudocódigo em português e Java.

Comandos em português	Equivalente em Java
`Escreva()`	`System.out.print()`
`escreva pulando linha()`	`System.out.println()`
`leia real`	`scanner.nextFloat()`
`leia inteiro`	`scanner.nextInt()`
`se condição então faça {   }`	`if (condição) {   }`

Este livro traz exemplos e trechos de código (*snippets*) principalmente em pseudocódigo, além de algumas linguagens populares, podendo ser usados diretamente em Java, C e Python ou, com pouco esforço, traduzidos para Fortran, Pascal, Pearl, PHP, Matlab, entre outras. Para os futuros programadores, recomendamos a instalação de um Ambiente de Desenvolvimento apropriado para a linguagem escolhida. A seção "Ambientes de desenvolvimento integrados" traz algumas opções de IDE disponíveis.

Para fazer um programa em Java é necessário declarar uma classe (programa) e nele um método principal (`main`), que será executado imediatamente ao iniciar-se a execução do programa. Em adição, é preciso importar outras classes (bibliotecas) contendo métodos que serão usados para entrada e saída de dados. Como resultado, a versão em Java do programa deve obrigatoriamente incorporar elementos de **Programação Orientada a Objetos** (POO). Embora não seja necessário o entendimento do paradigma para aplicação deste livro, o Capítulo 7 – "Tópicos avançados" traz detalhes sobre a estrutura de classes e métodos em linguagens desta categoria.

Código 1.18. Java: Média de 3 notas[10].

```
package livro_PI_cap1;
import java.util.Scanner;
public class Media {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Digite a 1a nota:");
    float nota1 = scanner.nextFloat();
    System.out.print("Digite a 2a nota:");
    float nota2 = scanner.nextFloat();
    System.out.print("Digite a 3a nota:");
    float nota3 = scanner.nextFloat();
    float media = (nota1 + nota2 + nota3) / 3f;
    System.out.print(Média=" + media);
  }
}
```

[10] Teste os códigos Java usando o console ou uma IDE de sua preferência. Existem também opções para testar *online*, como exemplo, veja http://www.browxy.com.

Outras características de cada linguagem incluem os métodos para entrada e saída de dados. No exemplo em Java a leitura dos dados é feita pela classe `Scanner`, (biblioteca de aquisição de dados) usando o dispositivo de entrada padrão do sistema (`System.in`), no caso o teclado. A saída padrão (`System.out`) é o console (*shell* ou *prompt* de comando, dependendo do sistema operacional). A classe `scanner` é também usada para leitura e gravação de arquivos em disco.

Em geral, nas linguagens orientadas a objetos, existe uma declaração de classe que identifica o nome do programa (no caso, `Media`). No Java, o nome do arquivo deve ser o mesmo da classe (No caso, `Media.java`). A classe deve obrigatoriamente conter um **método principal** `main` (código principal), o código a ser executado **primeiro** ao se rodar o programa. Os caracteres '{' e '}' denotam blocos de código, isto indica que o método `main` contém todos os comandos entre estes dois caracteres, enquanto a classe `Media` contém apenas o método `main`. Note também a tabulação das linhas, que visa auxiliar na compreensão da estrutura do código. Cumpridas estas formalidades, basta inserir o código apropriado no método principal (`main`) e executar o programa.

No seguinte exemplo, em C, é usada (importada) a biblioteca `stdio.h`, com os comandos `printf` e `scanf`, para realizar a escrita e leitura de dados, respectivamente.

Código 1.19. Exemplo C/C++ Exemplo de código[11].

```
#include "stdio.h"
int main(void) {
   float nota1, nota2, nota3, media;
   printf("Digite a 1a nota:");
   scanf("%f", &nota1);
   printf("Digite a 2a nota:");
   scanf("%f", &nota2);
   printf("Digite a 3a nota:");
   scanf("%f", &nota3);
   media = (nota1 + nota2 + nota3)/3;
   printf("Média= %f", media);
   return 0;
}
```

O uso eficaz de cada linguagem depende apenas da familiaridade do programador com a sintaxe, semântica e bibliotecas disponíveis, o que **só pode ser atingido com a prática**. Por exemplo, a programação em JavaScript envolve conhecimento prévio da estrutura HTML, da qual faz parte. Java requer consultas frequentes à documentação (*JavaDoc online*) visando localizar as funcionalidades desejadas dentro das bibliotecas incluídas com a distribuição.

Adicionalmente, para se incorporar inteligência aos programas, é necessário conhecimento de **lógica de programação** para saber como estruturar e organizar os

[11] Compiladores C e C++ estão disponíveis para todos os sistemas operacionais, embora seja recomendado o uso de alguma solução do tipo IDE, como Eclipse (http://eclipse.org), NetBeans (https://netbeans.org/) ou Visual Studio (https://www.visualstudio.com/), disponíveis gratuitamente.

programas visando criar um fluxo contínuo de código, partindo da **entrada (ou coleta) de dados**, seguido pelo **processamento da informação**, até a **saída de dados**, com a exibição dos resultados do processamento para o usuário.

ENTRADA DE DADOS → PROCESSAMENTO DA INFORMAÇÃO → SAÍDA

Para se definir a lógica de um programa, no entanto, muitos preferem abordar os três processos acima na seguinte ordem:

- 1ª questão: Qual a **informação de saída** desejada?
  (o que faz o programa)
- 2ª questão: Qual a **entrada de dados** necessária para obtê-la?
  (o que deve ser fornecido ao programa)
- 3ª questão: Qual o **processamento**?
  (procedimento para **transformar** a entrada: **dados**, na saída: **informação**)

As estruturas fundamentais de lógica de programação, usadas para orientar o fluxo do processamento, comuns a todas as linguagens de programação, são apresentadas nos próximos três capítulos.

EXERCÍCIOS

1. Quais os elementos funcionais da arquitetura proposta por Von Newmann?
2. Classifique os dispositivos como de entrada [i]nput ou saída [o]utput de dados.
    a. Impressora
    b. Mouse
    c. Teclados
    d. Alto-falante
    e. Tela sensível ao toque
    f. Microsoft Kinect
    g. Web-camera
    h. Joystick
3. Defina em suas próprias palavras o que é um algoritmo.
4. Quais as 4 estruturas fundamentais presentes em praticamente todas as linguagens de programação?
5. Quais os tipos mais comuns de erro de programação?
6. Escreva um trecho de programa para trocar o valor de duas variáveis pré-existentes: A e B. No final da execução, A deverá ter o valor de B e B o de A.
7. Dado o código abaixo:

```
int A = 9;
int B = 2;
int C = 5;
```

Adicione linhas de código para reordenar os valores nas variáveis A, B e C em ordem crescente de A a C. Ao final da execução, `A` deverá contar o menor valor, `B` o valor médio e `C` o maior. Implemente em qualquer linguagem formal e imprima as variáveis na saída.

8. Traduza o seguinte trecho de pseudocódigo para Java:

```
real celsius, fahrenheit, kelvin;
celsius = leia("Qual a temperatura em celsius? ");
fahrenheit = celsius * 1.8 + 32;
kelvin = celsius + 273.15;
escreva("Celsius = ", celsius);
escreva("Fahrenheit = ", fahrenheit);
escreva("Kelvin = ", kelvin);
```

9. Complete o teste de mesa para o programa a seguir, linha a linha e compare os valores obtidos na execução simulada com os valores finais das variáveis.

	Programa Soma	Soma	A	B	C
1	Soma = 0				
2	A = 10				
3	B = A % 3				
4	Soma = Soma + B + 1				
5	B = Soma + A				
6	C = 1*1<<1+1				
	Valores finais →	**2**	**10**	**12**	**4**

10. Pesquise sobre como é feita a conversão de valores e variáveis de inteiro para real e vice-versa em Java, C e C++.

Capítulo 2

ORGANIZAÇÃO DE CÓDIGO



2 ORGANIZAÇÃO DE CÓDIGO

Programas sequenciais

Comentários

Desvios de fluxo

Subprogramas

Bibliotecas de funções

Funções, métodos e modularização

Reaproveitamento e manutenção de código

Exercícios

PROGRAMAS SEQUENCIAIS

Como visto no Capítulo 1, na seção sobre “Algoritmos, fluxogramas e lógica de programação”, um programa consiste de uma sequência lógica de comandos e operações que, executadas em ordem, realizam uma determinada tarefa. Em geral, um programa processa dados de entrada de forma a obter o resultado desejado ou saída.

A entrada e saída de dados são, comumente, realizadas utilizando os dispositivos de entrada e saída, como teclado e monitor, embora os dados requeridos para o processamento possam estar já contidos no código do programa, como mostrado no exemplo a seguir.

Código 2.1. Exemplo em Java.

```
// Entrada de dados
float x = 2, y = -1;
// Processamento
y = y - 2 * x * x - 4 * x + 1;
// Saída
System.out.println(y);
```

No caso acima, para se obter uma saída para um conjunto diferente de dados de entrada, é necessário modificar o código fonte e recompilar o programa, o que nem sempre é conveniente. Outra opção é realizar a entrada usando o teclado em um trecho do programa onde é feita a entrada de dados.

Como visto anteriormente, a sequência de operações com dados em um programa de computador é dividida em:

Entrada → Processamento → Saída

Esta estrutura pode ser observada em um programa qualquer, onde o processo é dividido entre as atividades acima mencionadas, por partes.

Código 2.2. Pseudocódigo – Partes do programa “gorjeta”.

```
#1 Entrada de dados:
#1.1 Definição das variáveis do programa:
Real: conta, gorjeta;

#1.2 Entrada de dados
conta = leia("Entre o valor da conta: ");

#2 Processamento:
gorjeta = conta * 0.2;

#3 Saída de dados (na tela):
escreva("Valor da gorjeta = " + gorjeta);
```

Ao se escrever um programa, primeiro devemos determinar o **propósito do programa** ou qual a **saída desejada**. Em seguida é levantado o **conjunto de dados de entrada** necessários para se obter tal saída. Por fim, elaboramos o **processamento** ou o procedimento (ordem das operações) para obter-se a saída desejada a partir dos dados de entrada. Logo, o procedimento para especificar um programa é definir, nesta ordem:

1. A **saída** desejada do programa
2. Os dados de **entrada** necessários (insumos)
3. O **processamento** ou **transformação dos dados** de entrada em saída

A linha de execução dos programas (*thread*) nem sempre é linear, podendo conter descontinuidades e desvios na execução, como será demonstrado a seguir.

COMENTÁRIOS

Comentários são partes do código usados pelo desenvolvedor para deixar **notas**. Quando definido um comentário, é dada uma instrução direta ao compilador/interpretador para ignorar a parte comentada. Isto quer dizer que os comentários **não serão considerados** quando o código for executado, logo, os comentários não são parte da execução. Cada linguagem tem sua própria maneira de introduzir comentários no código.

Código 2.3. Exemplo Java / JavaScript / C / C++ / C# / Objective-C / outras[12].

```
// Isto é um comentário de uma linha, a instrução '//' diz
// ao compilador para ignorar tudo até o final da linha.

float x = 2, y = 1;
z = y - 2 * x * x - 4 * x + 1;

/*
   Isto é um comentário de múltiplas linhas:
   a instrução '/*' diz ao compilador para ignorar tudo até
   encontrar um  */   z = 0;

// neste exemplo, apenas as partes marcadas seriam executadas
```

Os caracteres usados para comentar partes do código mudam de linguagem para linguagem. A tabela seguinte mostra os caracteres para comentários em algumas delas.

Tabela 2-1. Identificadores de comentário.

Linguagem	Comentário de uma linha	Comentário de várias linhas
Java / JavaScript / C / C++	// ...	/* .... */
Python	# ...	""" .... """
Matlab	% ...	%{ .... %}
Pascal	{ ... }	{* .... *}
SQL	-- ...	/* .... */

[12] Algumas outras linguagens que utilizam a notação '//' e '/* */' para comentários são: D, Go, PHP, PL/I, Rust, Scala, SASS, SQL, Swift, Visual Prolog, CSS e ActionScript.

DESVIOS DE FLUXO

Um programa consiste em uma sequência de comandos executados em ordem, em uma linha contínua de execução. No entanto, esta linha (em inglês: ***thread***) pode conter desvios ou descontinuidades. Estas descontinuidades podem enviar a execução para sub-rotinas ou mesmo para outros programas, que realizam parte do processamento, executam operações e podem devolver dados de saída, em resposta a parâmetros de entrada. Um exemplo de tal chamada seria:

Código 2.4. Exemplo de chamada de função: DOS / bash / shell / cli /etc.

```
copy database.xml database.bak
```

No exemplo acima, a função a ser usada é o comando `copy`, enquanto os nomes dos arquivos (origem e destino) são chamados **parâmetros**, dados necessários para dizer ao programa qual a entrada e a saída. O mesmo ocorre em linguagens funcionais:

Código 2.5. Exemplo de chamada de função (múltiplas linguagens).

```
copyFile("database.xml", "database.bak")
```

Aqui os dois parâmetros estão entre parênteses, com valores do tipo cadeia de caracteres (`String`) definindo os dados a serem operados. Finalmente, uma função pode retornar valores para a posição de onde foi chamada, de forma que depois de executada a função um valor será produzido, devendo ser atribuído a uma variável.

Código 2.6. Exemplo de chamada de função (múltiplas linguagens).

```
M = maximo(V1, V2, V3, V4, V5);
```

No exemplo acima, a função `maximo` recebe como parâmetros um conjunto de números (`Vn`) e retorna para a **posição de chamada** o valor do maior entre eles. Tal valor é armazenado na variável local `M`.

Estes métodos de chamada desviam o fluxo de execução para um **subprograma**, que realiza sua tarefa específica e, ao terminar, retorna o fluxo de execução para a próxima instrução depois da chamada de função.

PROGRAMAS E SUBPROGRAMAS

Em grande parte das linguagens de programação, o código dos programas pode ser dividido em um ou mais arquivos ou partes, cada parte contendo uma sequência de comandos com um objetivo, realizando uma tarefa dentro do programa. Dentro de um mesmo programa podem existir subprogramas com funções específicas ou subconjuntos de comandos que só serão executados em condições especiais.

Entre os subprogramas do programa principal, estão aqueles que só serão executados dada uma certa condição, como será visto no próximo capítulo "Desvios condicionais", como também trechos de código que são repetidos várias vezes, enquanto uma dada condição for verdadeira, tema do capítulo seguinte: Estruturas de repetição (laços).

Todas as linguagens vêm acompanhadas de **bibliotecas**, estas contendo funções ou programas de uso comum. São exemplos as funções para **cálculos matemáticos**, para **operações de entrada e saída**, para comunicação e conversão de dados. As funções matemáticas, como seno, cosseno, tangente, logaritmo, absoluto e raiz, estão presentes na biblioteca de funções matemáticas. Estas funções nada mais são que subprogramas, que recebem dados, os processam e retornam a saída ao local de chamada da função. Seguem alguns exemplos da biblioteca de funções matemáticas do Java:

Código 2.7. Biblioteca Java *Math*.

```
double r = 5, diametro, Area_circulo, Vol_esfera, teta, moeda;
diametro = 2 * r;
Area_circulo = Math.PI * Math.pow(r, 2); // pi . r²
Vol_esfera = (4.0/3.0) * Math.PI * Math.pow(r, 3);
teta = Math.atan(r); // arco tangente
moeda = Math.round(Math.random());  // 0 para cara, 1 para coroa
System.out.print("V=" + Vol_esfera + "\n Moeda=" + moeda);
```

A classe `Math` é um programa do tipo biblioteca, que contém vários métodos ou funções, no caso matemáticas, que facilitam ao programador fazer cálculos comuns sem ter que programá-los. A forma de utilizar estas funções é através da passagem de parâmetros: os **dados de entrada** do subprograma. No exemplo `Math.pow`, os parâmetros (argumentos) da função são a base e o expoente, onde o valor retornado é $base^{expoente}$. `Math.PI` é uma constante pré-programada na classe, que contém o valor de π com alta precisão. Funções podem aceitar **vários parâmetros**, como `Math.pow(b,x)` apenas um parâmetro, como em `Math.sin(x)`, `Math.cos(x)`, `Math.tan(x)`, `Math.atan(x)`, `Math.round(x)`, ou mesmo nenhum parâmetro, como é o caso em `Math.random()`, que retorna um valor real aleatório no intervalo de 0 a 1. Outro exemplo, a função `print()` (comando imprimir) está presente na maioria das linguagens e recebe como parâmetro uma cadeia de caracteres. A saída da função é a cadeia de caracteres na saída padrão do sistema, sem retorno ao ponto de chamada.

Ainda, é possível se programar operações que não existem em bibliotecas providas pela linguagem escrevendo suas próprias funções, bastando para isso introduzir o código apropriado para o processamento dos dados de entrada em saída em um subprograma.

FUNÇOES OU MÉTODOS DE USUÁRIO

O uso de funções facilita a reutilização de código, dado que uma função é um programa autocontido com entrada, processamento e saída, podendo ser copiado de um programa para o outro ou incorporado em uma biblioteca escrita pelo usuário. A seguir temos um exemplo de função de usuário:

Código 2.8. Pseudocódigo: exemplo de função.

```
função delta(recebe: real a, real b, real c) retorna real d {
     d = b² – 4ac
     retorne d;
}
```

Código 2.9. Pseudocódigo: programa completo usando a função.

```
principal {
     real valor = delta(5, -2, 4)
     escreva("O delta de 5x² – 2x + 4 é ” + valor)
}
função delta(recebe: real a, real b, real c) retorna real d {
     d = b² – 4ac
     retorne d
}
```

Código 2.10. Python: exemplo de função.

```
# a função é definida aqui
def delta( a, b, c ):
        d = b * b – 4 * a * c;
        return d;

# agora já pode ser usada
valor = delta( 5, -2, 4)
print "O delta de 5x^2 – 2x + 4 é ", valor
```

Código 2.11. Java: exemplo de função (método).

```
public class Ex_Metodo {

    public static void main(String[] args) {
        System.out.println("Delta = " + delta(1.0, 3.0, 1.0));
    }

    public static double delta(double a, double b, double c){
        double d = b * b – 4 * a * c;
        return d;
    }
}
```

Note que nos exemplos, as **chaves** '{ }' delimitam um subprograma, que inicia na abertura de **bloco** '{' e termina em '}'. O subprograma `delta`, do tipo função (chamado de método em POO, como no Java), recebe 3 parâmetros, valores reais, nas variáveis do subprograma nomeadas `a`, `b` e `c`, processando e devolvendo o **valor** da variável `d` (delta) para o ponto de chamada da função, com o comando de retorno. Para obter o valor de delta, basta chamar a função no programa principal passando parâmetros adequados.

TABULAÇÃO

Um conceito muito importante em programação é a "**endentação"** (*indentation*) ou **tabulação** do código. Note que sempre que um bloco ou subconjunto de comandos é iniciado com '{' a tabulação é incrementada, enquanto quando um subconjunto de comandos se encerra com '}', a tabulação é recuada. Isto permite visualizar claramente quando **um grupo de comandos define um subprograma**. Linguagens como **Python** fazem da tabulação precisa um requerimento obrigatório para definição da **lógica de programação**.

Alguns editores de programa e a maioria das IDEs já fazem a tabulação de forma automática. Alguns tabulam códigos sem formato prévio. Por exemplo, o NetBeans possui um macro para tabular os códigos de acordo com a sua estrutura, que pode ser acessado pela opção do menu "Código-fonte: Formatar" ou pelo atalho "Control+Shift+F".

LINGUAGENS FUNCIONAIS

Linguagens funcionais são aquelas em que a **organização do código** é feita com uso de **funções**. Em oposição a implementação em linha de operações que resultam em alteração do estado das variáveis locais (**programação imperativa**) a programação funcional comunica dados entre subprogramas através da **passagem de parâmetros** e retorno de valor/valores ao local de chamada.

Este paradigma de programação permite a organização dos programas em várias unidades funcionais, cada uma responsável pelo cumprimento de uma tarefa específica. Isto também facilita o reaproveitamento de código, dado que um subprograma, por conter relação conhecida de dados de entrada e de saída, é um programa em si, completo e autocontido, logo pode ser copiado de um código para outro que necessite da mesma função. Este conceito é conhecido como **caixa-preta**. Nele, quem usa a função não precisa ter conhecimento do funcionamento do seu programa, apenas de como é feita a comunicação de dados, ou seja, sobre os dados de entrada enviados e de saída recebidos.

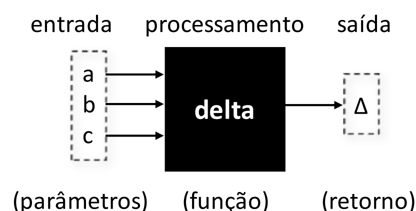


Figura 2.1. Subprograma (função) na visão "caixa-preta".

As funções auxiliares podem estar contidas no mesmo arquivo do programa principal ou em um arquivo separado, que agrupa funções do mesmo tipo na forma de uma **biblioteca de funções**. Neste caso, tal biblioteca deve ser **importada** com o comando apropriado, visando anexar o código com as funções ao programa principal.

Em linguagens orientadas a objeto, os **métodos** são subprogramas responsáveis por parte do processamento dos dados. Para programas sequenciais **estáticos** (com apenas uma única instância de execução), mesmo quando escritos em linguagens orientadas a objeto, não há qualquer diferença entre métodos e funções. Para maiores detalhes sobre o uso e funcionamento de métodos em classes de objetos, veja o Cap. 7 "Tópicos avançados".

Os métodos operam recebendo parâmetros de entrada e devolvendo, geralmente, um único dado como saída. O valor de saída, por sua vez, tomará lugar no ponto de chamada do método após sua execução. Um possível uso de métodos é para simplificar a escrita de código, criando atalhos para funções mais complexas, envolvendo múltiplos passos, assim reduzindo o tamanho do código principal e facilitando a compreensão, manutenção e reuso do código. Se quiséssemos, por exemplo, usar entrada e saída de dados no Java usando, ao invés da sintaxe padrão, os comandos `leia()` e `escreva()`, como feito nos exemplos em pseudocódigo, poderíamos fazê-lo criando métodos para traduzir estes comandos da sintaxe e semântica do pseudocódigo para Java.

Código 2.12. Pseudocódigo: conversão de temperatura.

```
principal {
        real celsius, fahrenheit, kelvin;
        celsius = leia("Qual a temperatura em celsius? ");
        fahrenheit = celsius * 1.8 + 32;
        kelvin = celsius + 273.15;
        escreva("Celsius = " + celsius);
        escreva("Fahrenheit = " + fahrenheit);
        escreva("Kelvin = " + kelvin);
}
```

Código 2.13. Java: métodos escreva texto e leia número real.

```
import java.util.*; // importa classes da biblioteca util
public class Temperatura_com_metodos {

    public static void main(String[] args) {
        float celsius, fahrenheit, kelvin;
        celsius = leia("Qual a temperatura em celsius? ");
        fahrenheit = celsius * 1.8f + 32f;
        kelvin = celsius + 273.15f;
        escreva("Celsius = " + celsius);
        escreva("Fahrenheit = " + fahrenheit);
        escreva("Kelvin = " + kelvin);
    }
    public static void escreva(Object texto) {
        System.out.println(texto);
    }
    public static float leia(Object texto) {
        Scanner teclado = new Scanner(System.in);
        System.out.print(texto);
        float valor = teclado.nextFloat();
        return valor;
    }

}
```

O programa em Java traz além do método principal `main()`, executado ao se iniciar o programa, dois métodos auxiliares: `escreva()` e `leia()` para números reais. Como optamos por números do tipo `float`, adicionamos a todos um 'f' para denotar precisão simples.

Código 2.14. Java: métodos escreva texto e leia `String`.

```
import java.util.*;  // importa classes da biblioteca util
public class Dialogo {

    public static void main(String[] args) {
        String nome = leia("Qual é o seu nome? ");
        escreva("Seja bem vindo, " + nome + "!");
    }

    public static void escreva(Object texto) {
        System.out.println(texto);
    }

    public static String leia(Object texto) {
        Scanner teclado = new Scanner(System.in);
        System.out.print(texto);
        String valor = teclado.nextLine();
        return valor;
    }
}
```

Em ambos os exemplos, o método `leia()` recebe um objeto qualquer contendo a mensagem a ser exibida, descrevendo o dado que deve ser entrado e a exibe. Em seguida, aguarda a entrada pelo usuário, lendo dado do teclado usando um objeto do tipo `Scanner`, e retorna um valor do tipo `String` ao ponto de chamada. No ponto de chamada, por sua vez, o valor retornado pelo método deverá ser recebido em uma **variável local** no método principal. Para ler um dado de outro tipo, o método deve ser modificado para lidar com o tipo de dado que deverá ser entrado pelo usuário.

O método `escreva()`, de forma semelhante, exibe na tela o objeto texto recebido como parâmetro, usando o comando apropriado do Java (`System.out.print` ou `println`). Neste caso, o método funciona meramente como um atalho, não retornando nada. O termo **`void`**, que denota "**vazio**", é usado quando não há valor a ser retornado do método. A saída produzida pelo método é apenas a sequência de caracteres exibidos na tela.

Repare que o código contido em ambos os métodos auxiliares não será executado a não ser que deliberadamente "chamado" a partir do método principal. Ao ser chamado, o programa é executado e o valor retornado, caso haja retorno.

Outra característica do Java é que a ordem dos métodos dentro de uma classe não importa, já que a execução é realizada através de chamadas. Em algumas linguagens a ordem ou posição das funções ou métodos importa. No C++, por exemplo, os métodos (ou funções) que aparecem depois do método principal devem ter sua assinatura declarada antes dele, como segue:

Código 2.15. C++: Exemplo completo de funções.

```
#include <iostream>  // importa biblioteca de entrada/saída
using namespace std;

void impar (int n);  // declaração da função impar
void par (int n);    // declaração da função par

int main () {
    int i;
    cout << "Entre um número: ";  // saída de dados na tela
    cin >> i;                     // entrada de dados do teclado
    if (impar(i))
        cout << "Numbero é impar.\n";
    if (par(i))
        cout << "Numbero é par.\n";
    return 0;
}
bool impar (int n) { // função impar recebe int retorna boleano
    if ((n%2)!=0)
        return true;
    else
        return false;
}
bool par (int n) {  // função par recebe int retorna boleano
    if ((n%2)==0)
        return true;
    else
        return false;
}
```

Note que as funções para verificar se o número é par ou ímpar utilizam um conceito que ainda não abordamos: condicionais (próximo capítulo). Usando o princípio da **caixa-preta**, apenas precisamos saber quais as entradas e saídas da função. No caso, ambas recebem um número inteiro e retornam um valor booleano, verdadeiro ou falso. Note que apenas o valor da variável é passado para o método. O nome dado a variável em cada função é indiferente do ponto de vista do compilador, visto que utilizam endereços de memória diferentes. O retorno, uma variável booleana, tem seu valor (verdadeiro ou falso) imediatamente usado em um desvio condicional, que imprime a mensagem apropriada.

O método principal em C/C++ deve retornar zero no caso de execução sem erros. Caso haja erro, será retornado -1 ou o código do erro ocorrido.

ESCOPO

Os subprogramas são programas independentes dentro do programa, logo possuem variáveis próprias para armazenar seus dados. Estas variáveis só existem no âmbito do subprograma e só durante cada execução (chamada) do mesmo, desaparecendo (ou sendo apagadas) ao termino do subprograma ou ao retornar em qualquer ponto com o comando `return`.

Quando são criadas variáveis dentro de um subprograma (inclui desvios, métodos e funções), estas variáveis serão consideradas **variáveis locais**, ou seja, pertencem apenas ao subprograma, não existindo fora dele. **Variáveis globais**, por outro lado, podem ser

criadas em um **escopo hierarquicamente superior** a todos os métodos, desta forma **permeando** todos os subprogramas. Logo, as variáveis globais têm escopo em todos os métodos.

Código 2.16. Pseudocódigo: Escopo

```
inteiro n = 1;  // variável global
principal {
    real b = 0;
    escreva("O delta de 5x² - 2x + 4 é ” + delta(5, -2, 4))
    n = n + 1
    escreva("n = " + n)
    escreva("b = " + b)
}
função delta(recebe: real a, b, c) retorna real d {
    d = b² - 4ac
    n = n + 1
    b = b + 1
    retorne d
}
```

No último exemplo, as variáveis `a`, `b`, `c` e `d` dentro da função `delta` são locais, isto é, só existem dentro deste subprograma e desaparecem quando ele termina. A variável `b` do programa principal se refere a outra posição de memória, logo, variáveis diferentes, em escopos diferentes, ocupando endereços distintos, mesmo quando compartilham um mesmo valor. A variável `n`, por outro lado, existe em todos os escopos, por ser uma **variável global**. Qualquer escopo pode acessá-la e modifica-la. Isto é particularmente útil quando se precisa retornar mais de um valor de uma função, já que o comando “retorne” retorna da função, por definição, apenas um valor.

Dado o escopo das variáveis `n` e `b`, a saída do programa acima seria:

```
O delta de 5x² - 2x + 4 é -76
n = 3
b = 0
```

Se pedíssemos para imprimir os valores de `c` ou `d` no método principal, obteríamos um erro, já que as variáveis não existem neste escopo.

REAPROVEITAMENTO E MANUTENÇÃO DE CÓDIGO

Outra vantagem do uso de funções e métodos, além da capacidade de se reaproveitar código já escrito em novos programas copiando os subprogramas desejados, é a possibilidade de se **atualizar** os métodos sem a necessidade de alterar o código do programa principal. Para tanto, basta que comunicação do método (**entradas** e **saídas**) permaneçam inalterados. Como exemplo, utilizamos um programa em Java com métodos para entrada e saída de dados com `leia()` e `escreva()`, baseado nos exemplos anteriores.

Código 2.17. Java: Diálogo usando o console.

```
import java.util.Scanner;  // importa apenas classe Scanner
public class Dialogo2 {

    public static void main(String[] args) {
        String nome = leiaTexto("Qual é o seu nome? ");
        escreva("Seja bem vindo, " + nome + "!");
        int idade = (int) leia("Qual a sua idade? ");
        escreva("Aposto que você nasceu em " + (ano()-idade));
    }

    public static int ano() {
        return Calendar.getInstance().get(Calendar.YEAR);
    }

    public static void escreva(Object texto) {
        System.out.println(texto);
    }

    public static String leiaTexto(Object texto) {
        Scanner teclado = new Scanner(System.in);
        System.out.print(texto);
        return teclado.nextLine();
    }

    public static float leia(Object texto) {
        Scanner teclado = new Scanner(System.in);
        System.out.print(texto);
        return teclado.nextFloat();
    }

}
```

No exemplo acima os métodos (em negrito) utilizam o console. Além dos métodos `ano()` e `escreva()`, utilizamos dois métodos para leitura: `leia()` e `leiaTexto()` para números reais e `Strings`, respectivamente. Repare também que o método `leia()` usado retorna um número real do tipo `float` e pode ser usado para ler qualquer formato numérico, porém, necessita de conversão para gravação do dado em variáveis de tipos

inteiros. No caso, para que possa ser armazenado na variável idade, esta conversão se faz adicionando o tipo da variável `(int)` na frente do número real[13].

```
C:\>javac Dialogo2.java
C:\>java Dialogo2
Qual é o seu nome? Pedro
Seja bem vindo, Pedro!
Qual a sua idade? 21
Aposto que você nasceu em 1995
```

Figura 2.2. Diálogos de entrada e saída pelo console.

Supondo que ao invés de fazer a entrada e saída de dados pelo console, desejemos utilizar janelas. A classe (biblioteca) `JOptionpane` proporciona esta funcionalidade através dos métodos `showMessageDialog()` e `showInputDialog()`. A alteração dos métodos pode ser feita sem a necessidade de alterar o código principal, já que as entradas e saídas permanecem inalteradas.

Código 2.18. Java: Diálogo usando janelas.

```
import java.util.*;  // importa a biblioteca util inteira
import javax.swing.JOptionPane; // importa apenas uma classe
public class Dialogo2_com_janelas {

    public static void main(String[] args) {
        String nome = leiaTexto("Qual é o seu nome? ");
        escreva("Seja bem vindo, " + nome + "!");
        int idade = (int) leia("Qual a sua idade? ");
        escreva("Aposto que você nasceu em " + (ano()-idade));
    }

    public static int ano() {
        return Calendar.getInstance().get(Calendar.YEAR);
    }

    public static void escreva(Object text) {
        JOptionPane.showMessageDialog(null,text);
    }

    public static float leia(Object text) {
        return Float.parseFloat(
                JOptionPane.showInputDialog(text)
        );
    }

    public static String leiaTexto(Object text) {
        return JOptionPane.showInputDialog(text);
    }

}
```

[13] *Type cast* é uma operação usada em algumas linguagens, como Java, C e C++, quando é necessário realizar conversões de dados que envolvem perda de precisão. Consiste em colocar o tipo desejado, entre parênteses, na frente do dado a ser convertido.

Ao executar o programa, note que o código principal permanece o mesmo, embora o comportamento da entrada e saída de dados realizada tenha mudado.

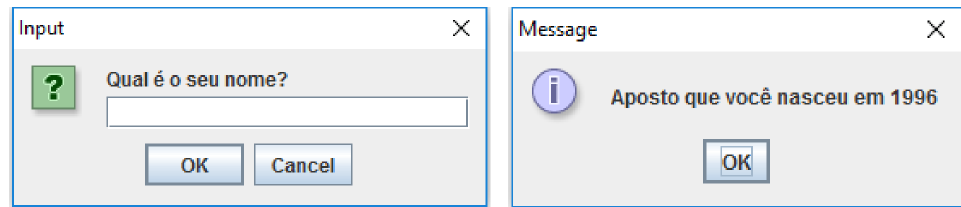


Figura 2.3. Diálogos de entrada e saída usando janelas.

EXERCÍCIOS

1. Crie um método que receba um valor real qualquer e retorne 0 se este valor for par ou 1 se o valor for ímpar (Dica: utilizar o operador resto %).

2. Descreva o procedimento ou função para receber um ponto em coordenadas cartesianas (X, Y) e retornar a distância até a origem (0, 0).

3. Crie um método para calcular o ângulo formado entre um par de pontos X, Y, e o eixo x no plano cartesiano.

4. Qual a saída do seguinte programa:

```
inteiro a = 1;
principal {
    inteiro b=2, c = 3
    (1)
    escreva("a = " + a)
    escreva("b = " + b)
    escreva("c = " + c)
    (2)
}
função incrementa(recebe: real a, b, c) retorna nada {
    a = a + 1
    b = b + 1
    c = c + 1
}
```

   a. Com as posições (1) e (2) em branco
   b. Quando o comando `incrementa(a, b, c)` está na posição (1)
   c. Quando o comando `incrementa(a, b, c)` está na posição (2)

5. Crie um método para calcular o ângulo formado entre um par de pontos X1, Y1 e X2, Y2 no plano cartesiano.

6. Crie uma função para converter de polegadas para centímetros.

7. Crie um programa com variáveis globais de um retângulo para `base`, `altura` e `área`. Crie no mesmo programa funções para calcular cada um dos 3 valores a partir dos outros 2: `calcula_base()`, `calcula_altura()` e `calcula_area()`.

8. O programa diálogo (Código 2.18) apresenta um ano de nascimento errado no caso da data ser anterior ao aniversário do usuário. Corrija este problema usando dados da data de nascimento do usuário e a data atual `Calendar.MONTH` e `Calendar.DATE`.

Capítulo 3

DESVIOS CONDICIONAIS





3 DESVIOS CONDICIONAIS

O que é um desvio condicional?

Condições com lógica booleana

Estrutura de um desvio condicional

Tipos de desvios condicionais

Encadeamento

Exercícios

O QUE É UM DESVIO CONDICIONAL?

O desvio condicional é a mais simples entre as estruturas lógicas não sequenciais em lógica de programação e fundamental para o entendimento de fluxo de código. A analogia básica com o processo de tomada de decisões ocorre quando imaginamos um cenário que proporciona duas possíveis alternativas de curso. Exemplos:

- Se **[condição]** então faça **[caminho caso verdadeiro]** senão **[caminho caso falso]**
- Se [está chovendo] então [resolver palavras cruzadas] senão [andar de bicicleta]
- Se [é quarta] então [comer feijoada]

O **fluxograma** ao lado descreve um programa para calcular a média de duas notas. Para tanto é feita a leitura de dois números, que devem ser entrados pelo usuário.

O paralelogramo, neste caso, indica entrada ou saída de dados.

A única operação efetuada é o cálculo da média, usando a fórmula da média para 2 números. A variável M **recebe** o valor calculado.

O losango indica a condição de desvio.

Note que para a condição lógica há apenas duas alternativas possíveis: **verdadeiro** ou **falso**.

Caso a condição seja verdadeira, o paralelogramo indica como saída a palavra “Aprovado”. Caso a condição retorne valor falso a saída será “Reprovado”.

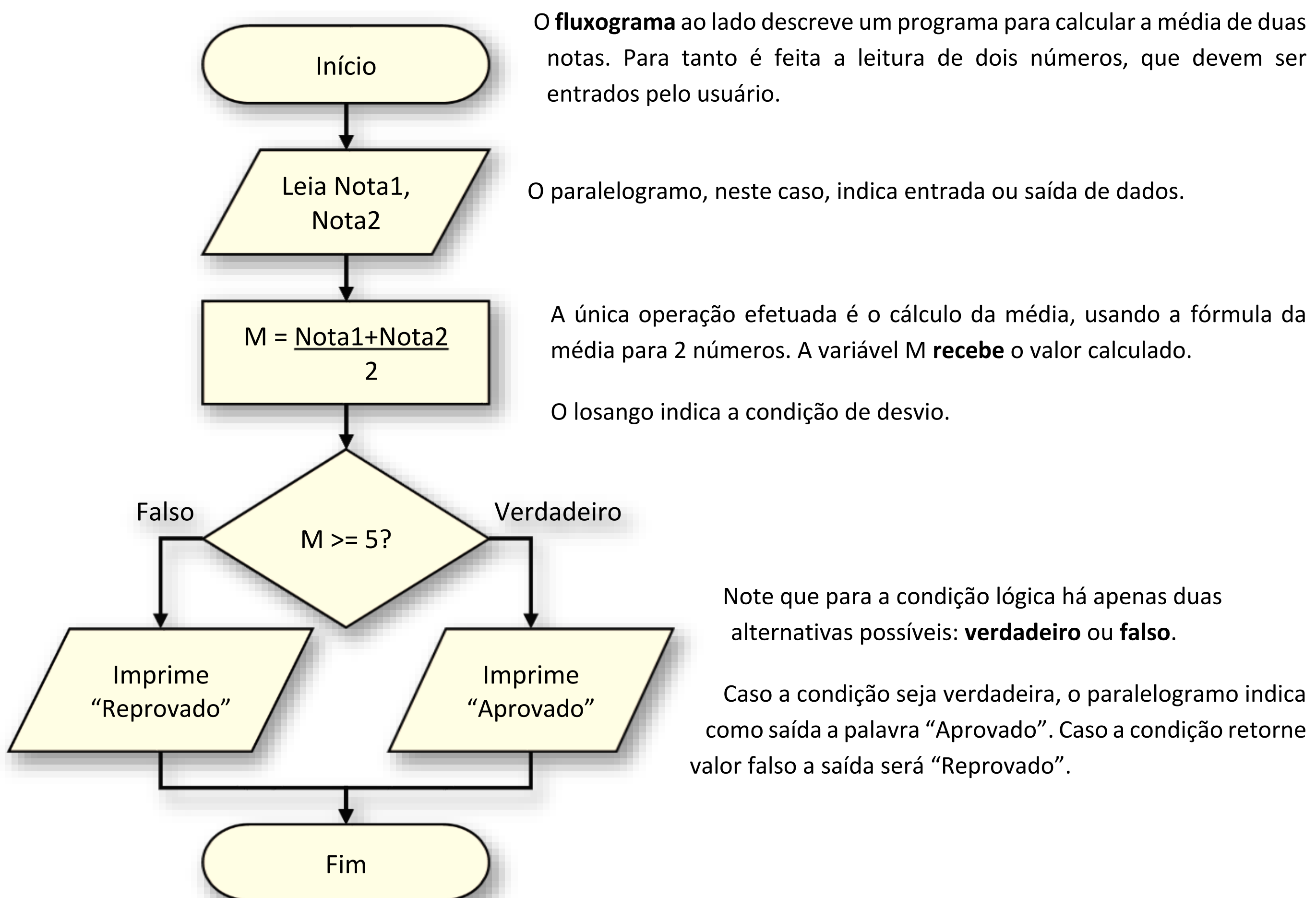


Figura 3.1. Exemplo de desvio condicional.

CONDIÇÕES COM LÓGICA BOOLEANA

O resultado de um teste condicional sempre resultará em um valor booleano, isto é, com dois resultados possíveis: verdadeiro (1) ou falso (0). Portanto, para condições, sempre usaremos combinações de operadores lógicos e relacionais para verificar o estado das variáveis verificadas. O seguinte pseudocódigo exemplifica algumas condições:

```
se vai chover, então leve um guarda-chuva.

se é feriado, então fique em casa.

se estou atrasado e está chovendo, então chame um taxi.

se minha nota é menor que 5, então fiquei de recuperação
```

Note que para todas as condições acima, a resposta para a condição é sempre: verdadeiro ou falso. Caso a condição seja verdadeira, será executada a operação ou operações especificadas na sequência. Codificando-se, as condições tomam a forma:

se (condição) [então faça] { comandos }

Código 3.1. Pseudocódigo: exemplos de condicionais.

```
se (condição) { comandos }

se (vai_chover) { leve um guarda-chuva }

se (feriado) ( fique em casa }

se (atrasado e chovendo) { chame um taxi }

se (nota<5) { imprima "ficou de ecuperação" }
```

Como variáveis booleanas só podem assumir o valor verdadeiro ou falso, **podem ser usadas diretamente nas condições**. Supondo que, por exemplo, "feriado" é uma variável do topo booleana, seu estado decide se o **subprograma** dentro das chaves será executado.

Quando trabalhamos com variáveis inteiras ou reais, no entanto, para verificar seu estado, precisamos empregar **operadores relacionais**.

- Operadores relacionais:
    - == (é igual a)
    - != (é diferente de)
    - > (é maior que)
    - < (é menor que)
    - >= (é maior ou igual a)
    - <= (é menor ou igual a)

Exemplos: Para A = 2, B = 5 e C = 10, qual o resultado (V ou F) das operações a seguir?

1. A < B
2. C == A * B
3. A + B >= C - A

Usando a precedência de operadores apresentada na subseção “Precedência de operadores” do Capítulo 1 (pag. 24), observamos que as operações algébricas são realizadas antes das relacionais, logo a resposta seria V, V e F.

Ainda, podemos combinar expressões relacionais usando os operadores lógicos:

- Operadores lógicos binários:
    - && (e), & (e bit-a-bit)
    - || (ou), | (ou bit a bit)
    - ! (não lógico ou complemento)
    - ~ (complemento bit-a-bit)
    - ^ (ou exclusivo ‘XOR’ bit-a-bit)
    - << N (*shift-left*, adiciona N zeros à direita do número em binário)
    - >> N (*shift-right*, elimina N dígitos a direita do número em binário)

Os operadores lógicos, como os relacionais, sempre resultarão verdadeiro ou falso, porém os operandos também são booleanos. Para as condições contendo AND, OR e XOR (ou exclusivo), as tabelas verdade para os dois operandos a esquerda e a direita, com valores lógicos representados na primeira linha e primeira coluna, são:

&&	V	F
V	V	F
F	F	F

||	V	F
V	V	V
F	V	F

^	V	F
V	F	V
F	V	F

Figura 3.2. Tabelas verdade para AND, OR e XOR.

Logo, os operandos devem ser ambos verdadeiros para que a operação E retorne verdadeiro, ao menos um deles verdadeiro para que o OU retorne verdadeiro e ambos diferentes para que o ou exclusivo retorne verdadeiro.

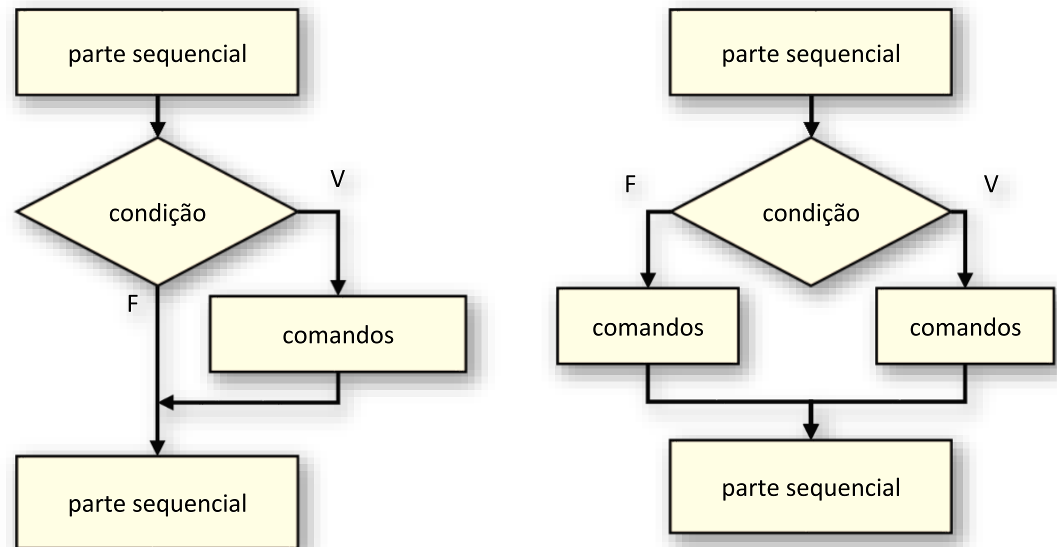
Estas expressões, quando combinadas, resultarão sempre em um valor booleano V ou F, que pode ser então introduzido em um desvio condicional visando a realização de um subprograma. Por exemplo, usando os valores X=1, Y=2 e Z=4, qual o resultado das expressões abaixo?

1. (X>0 && Y<2)
2. (Z>0 || Z<5 && Y==4)
3. (X>>1 ==0 || Y<<2>100)
4. (X!=0 && Y!=0 && Z<0)
5. (X=1)

Dada a **precedência de operadores** estudada anteriormente e a combinação apresentada acima, apenas as expressões 2 e 3 resultariam em verdadeiro, dado Z>0, assim como 1>>1 (*shift* à direita uma casa) em binário é 0, são ambas condições suficientes para que o resultado seja verdadeiro.

É importante ressaltar a diferença entre os operadores de **comparação** '==', com leitura '**é igual à**', e de **atribuição** '=', tendo a leitura '**recebe o valor de**'. Neste aspecto, a expressão 5 está incorreta, já que o operador de atribuição não faz sentido quando usado desta forma.

DESVIOS CONDICIONAIS SIMPLES E COMPOSTOS



Pseudocódigo

```
se (condição) então faça {
       Comandos
}
Volta para a parte sequencial
```

```
se (condição) então faça {
       Comandos
} senão faça {
       Comandos
}
Volta para a parte sequencial
```

C/C++/Java/JavaScript

```
if (condição) {
       Comandos*
}
```

```
if (condição) {
       Comandos*
} else {
       Comandos*
}
```

Matlab/Octav/SciLab

```
>>>if condição
         Comandos
   end
```

```
>>>if condição
          Comandos
      else
          Comandos
   end
```

Pyhton

```
if condição:
     comandos
parte sequencial**
```

```
if condição:
     comandos
else:
     comandos
```

Figura 3.3. Estruturas condicionais simples e compostas.

* Em C/C++/Java/ JavaScript, quando o desvio contém **apenas uma instrução** a ser executada, **as chaves podem ser omitidas**. Essa sintaxe também será utilizada nos pseudocódigos apresentados neste livro.

** Em Python, os blocos lógicos da condição dependem da tabulação ou endentação.

Digamos que, como exemplo, desejamos calcular as raízes da equação de segundo grau usando a função `delta()` introduzida no capítulo anterior. Sabemos que as raízes dependem do sinal do delta, logo, a solução de uma equação do segundo grau se dá resolvendo as seguintes condições:

1. Se $\Delta < 0$, x não possui raízes reais;
2. Se $\Delta = 0$, x possui duas raízes reais idênticas;
3. Se $\Delta > 0$, x possui duas raízes reais e distintas;
4. Calcule x usando a equação $x = -b \pm \sqrt{\Delta}/2a$.

Código 3.2. Pseudocódigo: raízes de uma equação de segundo grau.

```
principal {
    escreva("Resolve equação de 2° grau: ax² + bx + c")
    real a = leia("Entre com o primeiro termo 'a': ")
    real b = leia("Entre com o segundo  termo 'b': ")
    real c = leia("Entre com o terceiro termo 'c': ")
    real d = delta(5, -2, 4)
    escreva("O delta é " + valor)
    se (d < 0)  escreva("A equeção não possui raízes reais")
    se (d == 0) escreva("A raíz é " + (-a + raiz(d)/2*a))
    se (d > 0)  escreva("As raízes são " + (-a - raiz(d)/2*a)) +
                                   " e "  + (-a + raiz(d)/2*a))
}
função delta(real: a, b, c) retorna real:(b*b-4*a*c)
```

O pseudocódigo pode facilmente ser traduzido para Python, JavaScript ou Java:

Código 3.3. Python: exemplo de função.

```
def delta( a, b, c ):
    return b * b - 4 * a * c;
print "Resolve equação de 2° grau na forma ax^2 + bx + c"
a = float(raw_input("Entre com o primeiro termo a:"))
b = float(raw_input("Entre com o segundo  termo b:"))
c = float(raw_input("Entre com o terceiro termo c:"))
d = delta(a,b,c)
print "O delta é ", d
if d< 0: print "A equeção não possui raízes reais."
if d==0: print "A raíz é ", -a +d**(1/2.0) / 2*a
if d>0:  print "Raízes:",-a-d**(1/2.0)/2*a,",",-a+d**(1/2.0)/2*a
```

Código 3.4. JavaScript: raízes de uma equação de segundo grau.

```
    document.write("Resolve equação de 2° grau:<BR>");
    document.write("no formatto ax^2 + bx + c<BR>");
    a = parseFloat(prompt("Entre com o primeiro termo a: "));
    b = parseFloat(prompt("Entre com o segundo  termo b: "));
    c = parseFloat(prompt("Entre com o terceiro termo c: "));
    d = delta(a, b, c);
    var saida = "O delta é " + d + "<BR>";
    if (d < 0)  saida += "A equeção não possui raízes reais";
    if (d == 0) saida += "A raíz é " + (-a +Math.sqrt(d)/2*a);
    if (d > 0)  saida += "As raízes são " +
                     (-a -Math.sqrt(d)/2*a) + " e " +
                     (-a +Math.sqrt(d)/2*a);
    document.write(saida + "<BR>");
function delta(a, b, c) {    return b*b-4*a*c;    }
```

Código 3.5. Java: raízes de uma equação de segundo grau.

```
public class Equacao2oGrau {

    public static void main(String[] args) {
        escreva("Resolve equação de 2° grau: ax^2 + bx + c\n");
        double  a = leia("Entre com o primeiro termo 'a': "),
                b = leia("Entre com o segundo  termo 'b': "),
                c = leia("Entre com o terceiro termo 'c': "),
                d = delta(a, b, c);
        String res = "O delta é " + d + "\n";
        if (d < 0)  res += "A equeção não possui raízes reais";
        if (d == 0) res += "Raíz: " + (-a + Math.sqrt(d)/2*a);
        if (d > 0)  res += "Raízes: "
                    + (-a -Math.sqrt(d) / 2*a) + " e "
                    + (-a +Math.sqrt(d) / 2*a);
        escreva(res + ".\n"); // exibe sentença e pula linha
    }
    public static double delta(double a, double b, double c) {
        return b*b-4*a*c;
    }
    public static float leia(Object texto) {
        System.out.print(texto);
        return new java.util.Scanner(System.in).nextFloat();
    }
    static void escreva(Object o) {
        System.out.print(o);
    }
}
```

No programa, a condição determina qual caminho de execução seguir através de três desvios condicionais independentes. Note que a eficiência da sequência de condicionais poderia ser aumentada se introduzindo um comando “senão” (*else*) entre as condições, de forma que apenas um dos caminhos seja executado, sem a necessidade de perguntas adicionais após uma das condições ser atendida, como seria esperado para o caso. A próxima seção mostra como encadear diversos desvios condicionais em cascata desta forma.

Note que em Java, assim como em C, as sentenças são finalizadas com um **ponto e vírgula**, assim como é possível se declarar várias variáveis do mesmo tipo em uma única sentença, separando-as com vírgulas. Dado que, neste exemplo, cada desvio condicional realiza **um único comando**, as chaves { } definindo um bloco **podem ser omitidas**. Utilizando as funções introduzidas anteriormente e a classe `Math` (biblioteca de funções matemáticas) o código pode ser simplificado e resumido. Veja também que devido a execução na ordem de precedência dos operadores, é possível fazer o cálculo da raiz diretamente no texto da mensagem a ser exibida na saída, dispensando a criação de variáveis que não teriam mais uso após a exibição do valor.

DESVIOS CONDICIONAIS ENCADEADOS

O encadeamento pode ser feito introduzindo hierarquicamente outros desvios na condição verdadeira ou falsa de um desvio condicional. Considerando o primeiro exemplo deste capítulo, poderíamos encadear desvios para atribuir um conceito.

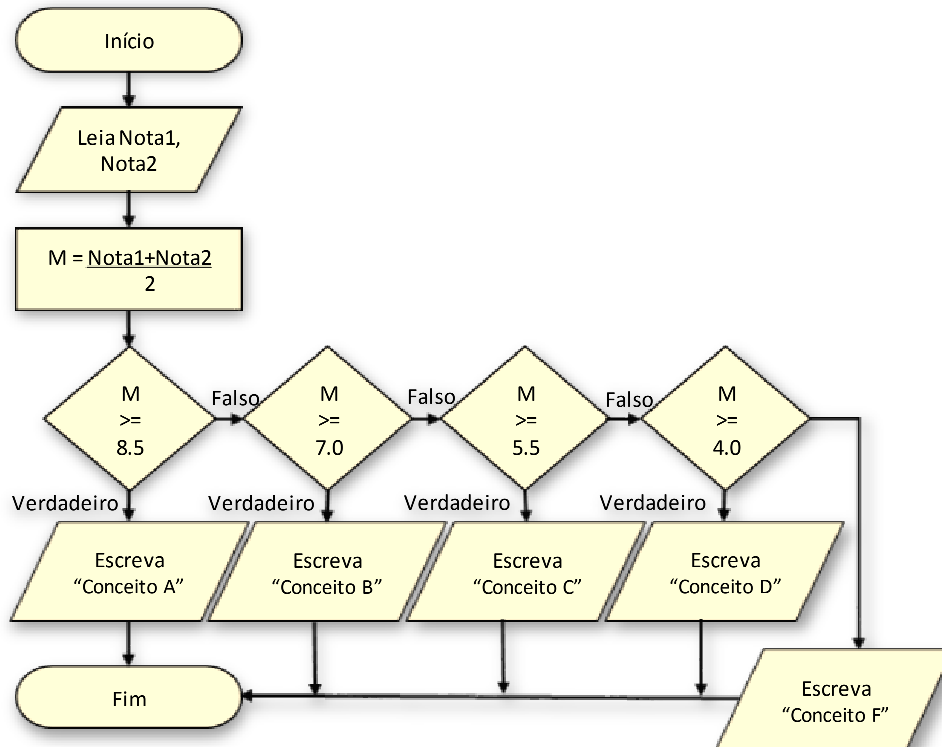


Figura 3.4. Desvios condicionais encadeados pela condição falsa.

Dado um desvio condicional, executamos o desvio seguinte apenas caso a condição seja falsa (ou verdadeira, dependendo da lógica adequada ao problema). Isso elimina a necessidade de combinar vários testes lógicos em cada desvio condicional, dado que já sabemos, ao avaliar a segunda condição, o resultado do teste anterior. No exemplo da Figura 3.4, a segunda condição é verificada apenas no caso a primeira ter retornado falso, logo, a média não pode ser maior ou igual à '8.5', o que já esta condição foi testada e resultou "falso" no desvio anterior.

A principal vantagem desta estrutura é sua eficiência, já que executamos os testes apenas até obter um resultado verdadeiro, pulando o restante dos testes quando a classificação da média for definida.

Traduzindo o fluxograma apresentado na figura para pseudocódigo, obtemos:

Código 3.6. Pseudocódigo: cálculo do conceito final.

```
nota1 = leia("Digite a 1a nota:");
nota2 = leia("Digite a 2a nota:");
media = (nota1 + nota2)/2;
se media >= 8.5 então
    escreva("Conceito A");
senão se media >= 7.0
    escreva("Conceito B");
senão se media >= 5.5
    escreva("Conceito C");
senão se media >= 4.0
    escreva("Conceito D");
senão
    escreva("Reprovado! Conceito F");
```

Código 3.7. JavaScript: cálculo do conceito final.

```
var nota1=parseFloat(prompt("Digite a 1a nota:"));
var nota2=parseFloat(prompt("Digite a 2a nota:"));
media = (nota1 + nota2)/2;
if (media >= 8.5)
    document.write ("Conceito A.");
else if (media >= 7.0)
    document.write ("Conceito B.");
else if (media >= 5.5)
    document.write ("Conceito C.");
else if (media >= 4.0)
    document.write ("Conceito D.");
else
    document.write ("Reprovado! Conceito F.");
```

Código 3.8. Java: cálculo do conceito final.

```
System.out.print ("Digite a 1a nota:");
float nota1 = scanner.nextFloat();
System.out.print ("Digite a 2a nota:");
float nota2 = scanner.nextFloat();
float media = (nota1 + nota2)/2;
if (media >= 8.5)
    System.out.print ("Conceito A\n");
else if (media >= 7.0)
    System.out.print ("Conceito B\n");
else if (media >= 5.5)
    System.out.print ("Conceito C\n");
else if (media >= 4.0)
    System.out.print ("Conceito D\n");
else
    System.out.print ("Reprovado! Conceito F.\n);
```

Código 3.9. C: cálculo do conceito final.

```
#include "stdio.h"
int main(void) {

    float nota1, nota2, media;
    printf("Digite a 1a nota:");
    scanf("%f", &nota1);
    printf("Digite a 2a nota:");
    scanf("%f", &nota2);
    media = (nota1 + nota2)/2;
    if (media >= 8.5)
        printf("Conceito A.\n");
    else if (media >= 7.0)
        printf("Conceito B.\n");
    else if (media >= 5.5)
        printf("Conceito C.\n");
    else if (media >= 4.0)
        printf("Conceito D.\n");
    else
        printf("Reprovado! Conceito F.\n);
    getch();
    return 0;
}
```

Código 3.10. Python: cálculo do conceito final.

```
nota1 = float(raw_input("Digite a 1a nota:"))
nota2 = float(raw_input("Digite a 2a nota:"))
media = (nota1 + nota2)/2
if media >= 8.5:
    print("Conceito A")
elif media >= 7.0:
    print ("Conceito B");
elif media >= 5.5:
    print ("Conceito C")
elif media >= 4.0:
    print ("Conceito D")
else:
    print ("Reprovado! Conceito F")
```

Outra possibilidade seria encadear o “senão” na condição verdadeira, para tanto, a lógica do programa deveria ser alterada:

Código 3.11. Pseudocódigo: cálculo do conceito final, com encadeamento em `true`.

```
nota1 = leia("Digite a 1a nota:");
nota2 = leia("Digite a 2a nota:");
media = (nota1 + nota2)/2;
se (media < 8.5)
    se (media < 7.0)
        se (media < 5.5)
            se (media < 4.0)
                escreva("Reprovado! Conceito F");
            senão
                escreva("Conceito D");
        senão
            escreva("Conceito C");
    senão
        escreva("Conceito B");
senão
    escreva("Conceito A");
```

OPERADOR CONDICIONAL

O operador condicional, presente em algumas linguagens como Java, C. C++, entre outras, é uma forma de reduzir a instrução se-então-senão quando o resultado é apenas um valor.

O formato do operador condicional '(? :)' é:

Var = (condição_booleana **?** valor_quando_verdadeiro **:** valor_quando_falso**)**

Uma forma condicional compacta, equivalente a estrutura:

```
 se (condição_booleana)
        Var = valor_quando_verdadeiro
senão
        Var = valor_quando_falso
```

Nos exemplos a seguir podemos ver a praticidade deste operador visando diminuir o número de comandos.

Código 3.12. Java, C, C++: uso do operador condicional compacto

```
float A = leia("Valor: ");"
A = (A>10 ? 10: A);    // limita o valor de A a até 10
A = (A<0  ?  0: A);    // A sempre > 0

media = A * 2;
System.out.println ("Você foi " +
                   ( media>=5 ? "aprovado!":"reprovado!") );

Boolean chuva = false;
String texto = ( !chuva ? "levante-se": "fique na cama");
System.out.println (texto); // Qual será a mensagem impressa?
```

Dada a condição, que pode ser uma variável booleana ou uma expressão lógica ou relacional, com resultado verdadeiro ou falso, se verdadeira, `A` recebe o valor 10, se falsa, `A` mantém o valor atual. De forma similar, a variável booleana `chuva` determina o valor da `String texto`, que recebe o valor "`levante`" caso a condição seja verdadeira e "`fique na cama`" caso esta seja falsa. No caso, a mensagem exibida será "`levante`" (note o operador de negação '`!`' antes da variável `chuva`).

EXERCÍCIOS

1. Utilizando apenas condicionais e o operador resto (%); a) Faça um programa para determinar se um número entrado pelo teclado é par ou ímpar, exibindo a mensagem apropriada na tela. b) Modifique o programa anterior para verificar se o número entrado é múltiplo de 3.

2. Faça um programa que leia (peça para o usuário digitar) três números inteiros quaisquer, armazenando nas variáveis `A, B` e `C` e imprima os números em ordem do menor para o maior.

3. Usando a Tabela 1-3. Precedência de Operadores e o operador condicional, faça o teste de mesa para descobrir o estado final de cada variável na execução do seguinte trecho de programa (descubra primeiro sem rodar o programa, então confira o seu resultado executando o código):

```
int x = 2, y = 4; x += 2 * (y++ / 3 +2);
float a = 10, b = 2, c = 3, d = 4;
a *= b + (++c + d) % b;
boolean f = x>y|a+b*c!=(a+b)*c&&a++<=20; d -= b++%--d;
int r = (f&&x>a?0:-1); c -=- --r;
```

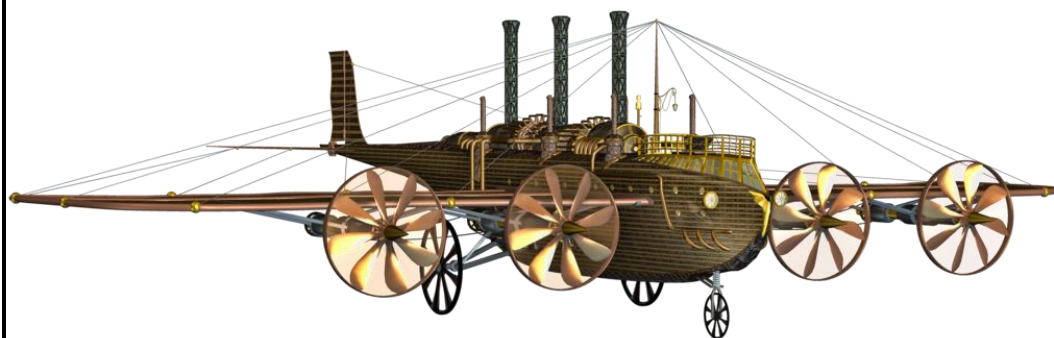
4. Faça um programa que receba 3 valores inteiros nas variáveis `A, B` e `C` e ordene os valores nas próprias variáveis, de forma que, no final da execução, a variável `A` contenha o menor valor e `C` o maior valor. O programa deve usar apenas 4 variáveis: `A, B, C` e `T`.

5. Faça um programa em qualquer linguagem para determinar a classificação do peso de um indivíduo, de acordo com a tabela:

Tabela 3-1. IMC = peso / altura^2	
Magro	IMC até 18,5
Saudável	IMC até 25,0
Acima do peso	IMC até 30,0
Obeso	IMC até 35,0
Morbidez	IMC 35 mais

Capítulo 4

LAÇOS



4 ESTRUTURAS DE REPETIÇÃO (LAÇOS)

Quando usar repetições?

Tipos de estruturas de repetição

Laços aninhados

Validação de dados com laços

Interrupção da execução dos laços

Exercícios

QUANDO USAR REPETIÇÕES?

As estruturas de repetição são recomendadas para quando um padrão de código é repetido várias vezes sequencialmente, apenas alterando-se o valor de uma ou mais variáveis entre os comandos repetidos. Veja no exemplo a seguir, quando queremos imprimir a tabuada de um número `t` entrado pelo usuário no formato:

```
Tabuada x (número de 1 a 10) = valor
```

```
Inteiro: t, n = 1;
t = leia("Qual a tabuada? ");
escreva(t + " x " + n + " = " + t * n);        n = n + 1;
escreva(t + " x " + n + " = " + t * n);        n = n + 1;
escreva(t + " x " + n + " = " + t * n);        n = n + 1;
escreva(t + " x " + n + " = " + t * n);        n = n + 1;
escreva(t + " x " + n + " = " + t * n);        n = n + 1;
escreva(t + " x " + n + " = " + t * n);        n = n + 1;
escreva(t + " x " + n + " = " + t * n);        n = n + 1;
escreva(t + " x " + n + " = " + t * n);        n = n + 1;
escreva(t + " x " + n + " = " + t * n);        n = n + 1;
escreva(t + " x " + n + " = " + t * n);        n = n + 1;
```

Embora o código não pareça extenso, é fácil imaginar uma situação onde tenhamos que repetir 100, 500, 1000 vezes a mesma operação. O exercício abaixo exemplifica este caso:

Exercício: Exiba os primeiros 100 números da sequência de Fibonacci. A sequência é dada pela série:

```
Sequência de Fibonacci = 0, 1, 1, 2, 3, 5, F(n-1) + F(n-2), ...
```

Claramente, devemos pensar em uma estrutura que permita computar os 100 elementos sem escrever 100 vezes o mesmo código.

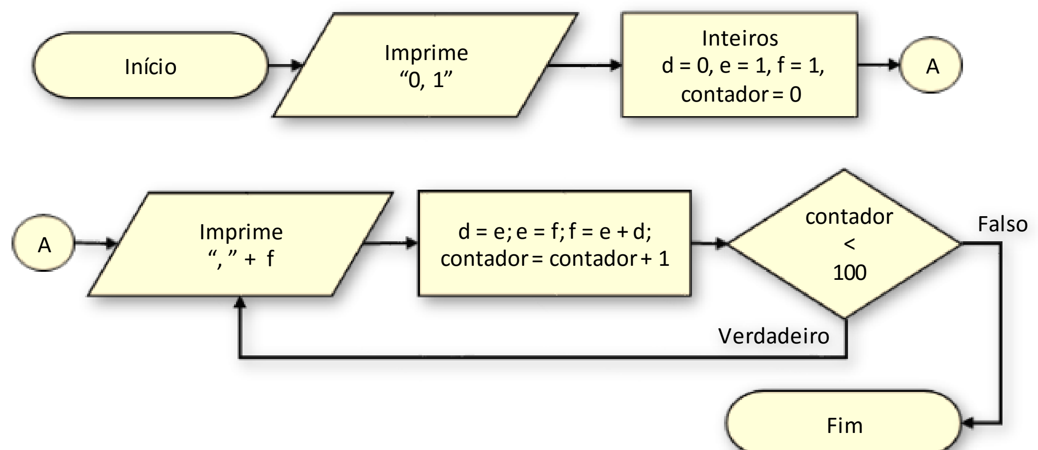
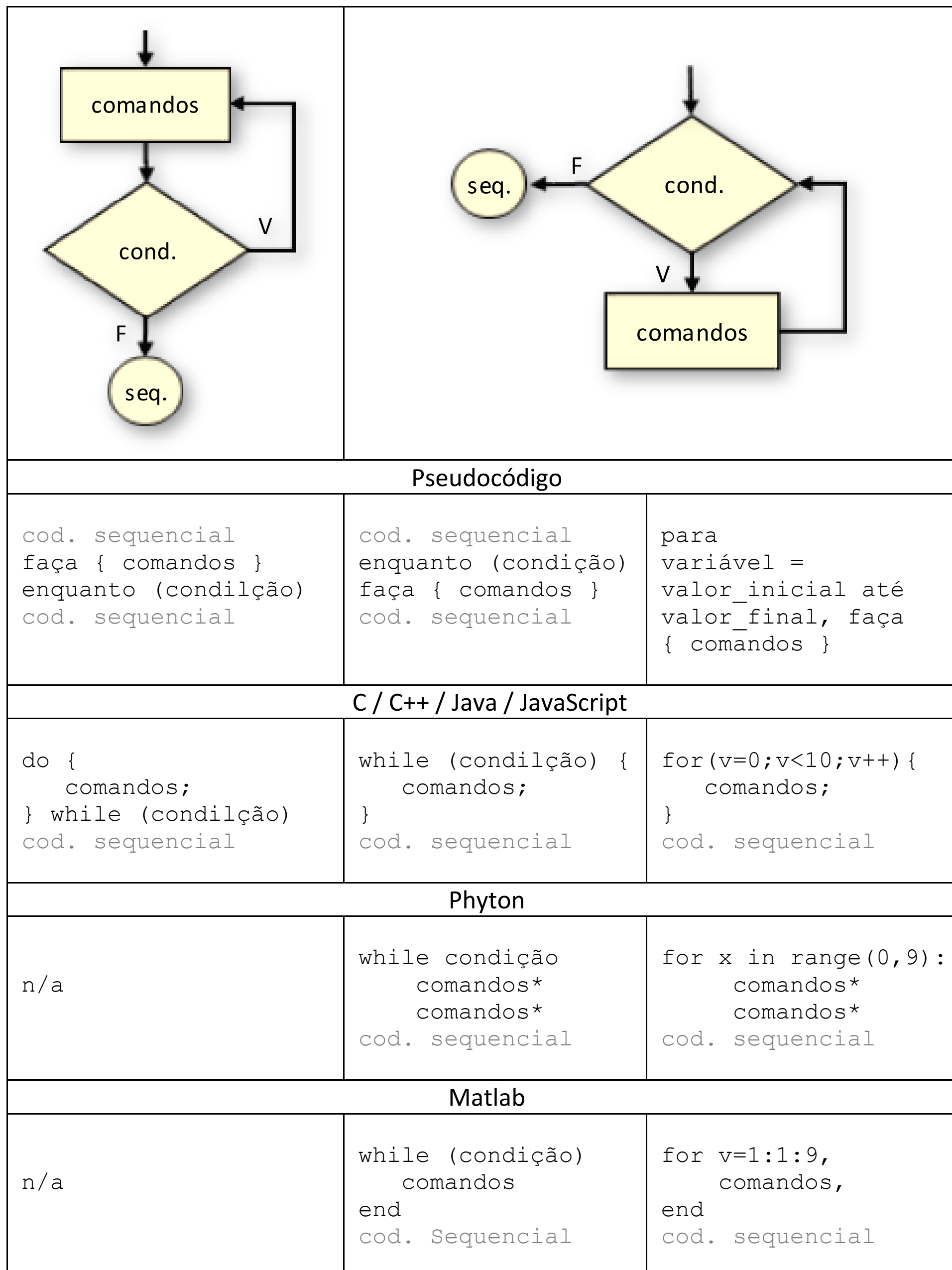


Figura 4.1. Fluxograma de cálculo da sequência de Fibonacci

TIPOS DE ESTRUTURAS DE REPETIÇÃO



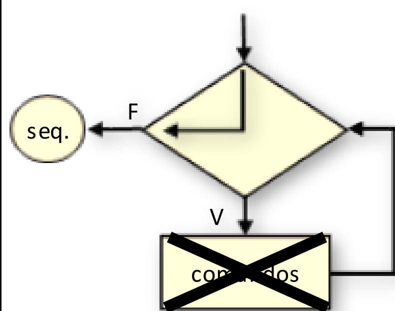
Pseudocódigo		
cod. sequencial faça { comandos } enquanto (condilção) cod. sequencial	cod. sequencial enquanto (condição) faça { comandos } cod. sequencial	para variável = valor_inicial até valor_final, faça { comandos }
C / C++ / Java / JavaScript		

do {
   comandos;
} while (condilção)
cod. sequencial
	while (condilção) {
	   comandos;
	}
	cod. sequencial
		for(v=0;v<10;v++){
		   comandos;
		}
		cod. sequencial

Phyton		
n/a	while condição comandos* comandos* cod. sequencial	for x in range(0,9): comandos* comandos* cod. sequencial
Matlab		
n/a	while (condição) comandos end cod. Sequencial	for v=1:1:9, comandos, end cod. sequencial

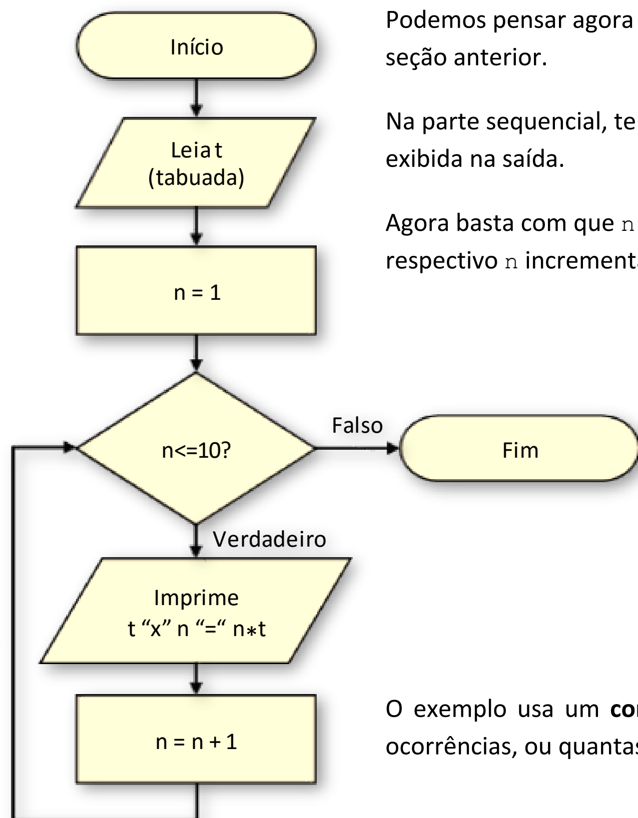
Figura 4.2. Exemplos de laços.

* Repare que Python utiliza as tabulações (endentações) para definir o escopo do laço. As linhas com tabulação incremental fazem parte do laço. Quando o recuo diminui significa o fim do laço, e o código a partir dali é sequencial.



As estruturas de repetição apresentadas diferem unicamente na ordem em que a verificação da condição ocorre na estrutura do laço, antes ou depois de executar o bloco de comandos. Note que no caso do `enquanto-faça` é necessário que a condição seja verdadeira para que os comandos presentes no bloco de execução sejam processados. Neste caso, se ao entrar no comando `enquanto` (`while`) a condição do teste for falsa, oposto ao faça-enquanto, o subprograma não será executado. Isto é, todo o código dentro do bloco do laço será pulado já na verificação inicial da condição, seguindo diretamente para a parte sequencial subsequente, similar ao que ocorre no `se-então`.

Figura 4.3. Caso no qual o enquanto-faça inicia com uma condição falsa.



Podemos pensar agora em uma possível estrutura do programa tabuada apresentado na seção anterior.

Na parte sequencial, temos a leitura do número `t` correspondente a tabuada a ser exibida na saída.

Agora basta com que `n` varie de 1 até 10 enquanto, exibimos cada linha com o respectivo `n` incrementado.

O exemplo usa um **contador**, ‘`n`’, ou seja, uma variável usada para “contar” quantas ocorrências, ou quantas vezes um código é executado.

Figura 4.4. Fluxograma do programa "tabuada".

A forma **faça-enquanto** é recomendada quando queremos que os comandos contidos no laço sejam **executados ao menos uma vez**, mesmo que a condição seja inicialmente falsa:

Código 4.1. Pseudocódigo: Exemplo laço faça-enquanto.

```
Real: nota, média, acumulador = 0,contador =0;
Caractere: resposta;

faça {
   nota = leia("Entre com uma nota: ");
   acumulador = acumulador + nota;
   contador = contador + 1;
   resposta = leia("Deseja continuar? (s/n): ");
} enquanto (resposta == 's')

média = acumulador / contador;
imprima ("A média das " + contador + " notas é " + média);
```

Além do **contador**, o programa usa um **acumulador** (variável que acumula). Repare que a condição `resposta == 's'` no **`faça-enquanto`** é falsa até que seja efetuada a leitura da variável `resposta` dentro do laço. Apenas no caso de o usuário entrar com o caractere `'s'`, o laço será repetido novamente. Isto quer dizer que o estado da condição é falso na primeira execução do código do laço.

Exercício: como ficaria o exemplo acima em Java?

Código 4.2. JavaScript: exemplo de laço while[14].

```
n = 1; t = parseInt(prompt("Qual a taboada? "));
while (n <= 10) {
    document.write (t + " x " + n + " = " + (t * n++) + "<BR>");
}
```

Note que no exemplo acima, se esquecermos de incrementar a variável n (comando `n++`) o valor permaneceria sempre igual a 1. Neste caso, a condição do laço seria para sempre verdadeira e a repetição jamais acabaria. Esta condição, onde a execução fica "presa" dentro do laço, é conhecida como **DEADLOCK**. *Deadlocks* geram erro de finalização de programa, que executará **eternamente**, podendo travar o programa, o teclado e mouse ou até mesmo o computador, neste caso, sendo necessário um RESET para sair do laço.

> Para evitar um **DEADLOCK** do laço, devemos sempre garantir que a condição de repetição eventualmente torne-se falsa, de forma que, em algum ponto da execução, o laço tenha um fim.

Para casos como o do exemplo "tabuada", onde conhecemos o valor inicial e final da repetição, a forma mais eficiente é o `para variável = valor_inicial até valor_final`. A vantagem deste método é já incluir um contador para a variável.

[14] Você pode testar este código no site de apoio ou em seu navegador, numa página HTML.

Código 4.3. Java: Exemplo de tabuada com laço `para`.

```
int t = (int) leia("Qual a taboada? ");
for (int n = 1; n <= 10; n++) {
    System.out.println(t + " x " + n + " = " + t * n);
}
```

Usando a função `leia()` para leitura de um inteiro (classe `Scanner` com um objeto instanciado, como apresentado no Código 2.13), o comando "`para n = 1 até 10 de 1 em 1`" cuida de assumir valores para `n` em todo o intervalo, executando os comandos delineados pelas chaves para cada valor. A estrutura poderia ser usada de forma idêntica no código em JavaScript.

O intervalo e passos podem ser ajustados como se desejar. Se, por exemplo, quiséssemos uma contagem regressiva, bastaria mudar a operação de incremento e os valores inicial e final.

Código 4.4. Java: Exemplo de laço `para`, em ordem reversa.

```
public static void main(String[] args)
                   throws InterruptedException {
                   // trata possível erro ao suspender a thread
   for (int n = 100; n > 0; n--) {
      System.out.println(n + " segundos para o lançamento.");
      Thread.sleep(1000); // dorme 1000 milissegundos (1s)
   }
   System.out.println("Ignição!");
}
```

No código acima, a mensagem é exibida a cada segundo, com valores de 100 a 1, terminando com a mensagem "Ignição! ". O comando `sleep` suspende a execução do programa por tempo determinado, deixando o processador livre para realizar outras tarefas.

Ainda é possível se definir um laço que conte em intervalos regulares ou irregulares.

Código 4.5. JavaScript: Exemplo de laço for, com passo diferente de 1.

```
document.write("Múltiplos de 11 até 1100: ");
for (n = 11; n <=1100; n = n + 11) {
    document.write("  " + n);
}
```

LAÇOS ANINHADOS

Uma estrutura muito usada em repetições são laços aninhados, quando para cada repetição de um procedimento temos que repetir outro um certo número de vezes. Isto é representado no fluxograma abaixo.

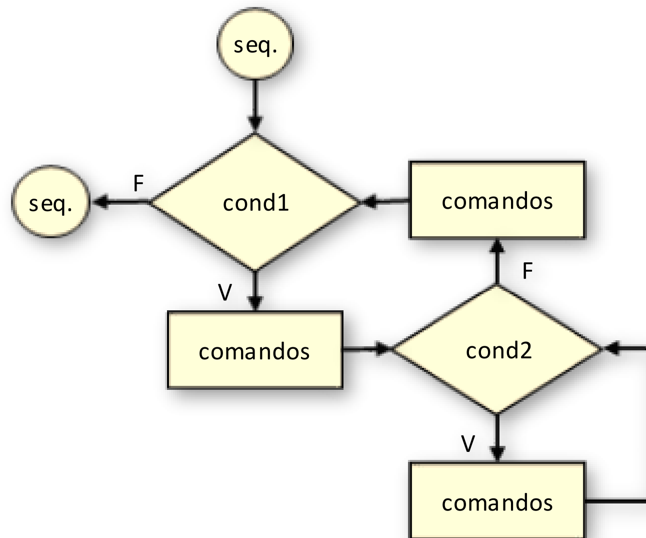


Figura 4.5. Laço aninhado.

Um laço aninhado define um subprograma dentro de um subprograma, ou seja, dentro de um laço podem existir comandos, condicionais ou mesmo outros laços, que chamados de aninhados. Nesta estrutura, podem haver comandos que são repetidos a cada iteração do laço principal ou a cada iteração do laço aninhado, dependendo do bloco em que se encontra.

Código 4.6. Java: Exemplo de laços aninhados.

```
for (int i = 1; i <= 4; i = i + 1) {   // Laço A
    System.out.print("A");
    for (int j = 1; j <= 8; j = j + 1) {   // Laço B
         System.out.print("B");
    }
    System.out.print("-");
}
```

No exemplo acima, para cada iteração do laço A o laço B será executado, repetindo seu subprograma 8 vezes para cada iteração do laço A. Isso quer dizer que o conteúdo do laço B será executado 32 vezes no total, ou seja, 4 iterações do laço A vezes 8 iterações do laço B. Logo, o ‘A’ será impresso apenas 4 vezes, enquanto o ‘B’, 32 vezes, gerando na saída:

```
ABBBBBBBB-ABBBBBBBB-ABBBBBBBB-ABBBBBBBB-
```

VALIDAÇÃO DE DADOS

Uma possível aplicação de laços é garantir que os dados entrados sejam válidos. **Validação de dados** é o nome dado a verificação dos valores de entrada, se os mesmos se encontram dentro dos limites previstos ou no formato adequado, notificando o usuário no caso de valores inválidos. O exemplo a seguir é a implementação em Java do Código 4.1, apresentado anteriormente, incorporando rotinas de validação de dados de entrada, indicando o erro e pedindo para o usuário entrar novamente o dado até que seja válido.

Código 4.7. Java: Exemplo de validação de dados.

```
public class Calculo_da_Media_Com_Validacao {
   public static void main(String[] args) {

      float nota = 0, media = 0, soma = 0; int i = 0;
      escreva("Entre com as notas + ENTER");
      do {
         boolean valida = false;  // variável de validação
         i++; // incrementa o contador
         do try {  //  tenta executar
               nota = leia("Entre com a nota " + i + ": ");
               if (nota < 0 || nota > 10)
                  escreva("Entre uma nota de 0 a 10!");
               else valida = true;
            } catch (Exception e) { //  executa no caso de falha
               escreva("Falha de digitação! Repita.");
            }
         } while (!valida); // repete enquanto entrada inválida
         soma += nota; // soma ao acumulador
      } while ( continua() );
      media = soma / i;
      escreva("A média das " + i + " notas é " + media);
   }
   public static boolean continua() { // Continua?
      escreva("Deseja continuar? (s/n): ");
      while (true) {       // repete até receber 's' ou 'n'
         String r = new java.util.Scanner(System.in).next();
         if (r.toLowerCase().charAt(0) == 's') return true;
         else if(r.toLowerCase().charAt(0) == 'n') return false;
         else escreva("Responda com 's' ou 'n': ");
      }
   }
   public static float leia(Object texto) {
      System.out.print(texto);
      return new java.util.Scanner(System.in).nextFloat();
   }
   static void escreva(Object o) {
      System.out.println(o);
   }
}
```

A estrutura de comando `try-catch` 'tenta' executar um bloco de código e captura uma exceção no caso de interrupção (falha), executando então um código alternativo para tratamento de erro. Note que os métodos (em negrito) já processam os dados e retornam valores apropriados para uso no ponto de chamada, quando presentes.

INTERRUPÇÃO DOS LAÇOS

Algumas linguagens permitem interromper a execução do laço através do comando 'interromper' ou 'quebrar' (*break*). Isto pode ser útil caso se queira interromper o laço em algum evento específico.

Código 4.8. Pseudocódigo: Exemplo de interrupção de laço.

```
real n = 100
enquanto (n>1) {
    d = leia ("Entre o divisor: ")
    se (d == 0)
        interromper
    n = n / d
    escreva(n)
}
```

O comando interromper para imediatamente a execução do laço em que se encontra, voltando a parte sequencial do código ou, no caso de laços aninhados, ao laço imediatamente superior hierarquicamente.

Código 4.9. Java: Exemplo de interrupção de laço.

```
public class Adivinha {
    public static void main(String[] args) {
        int n = (int) (Math.random() * 9 + 1); // de 1 a 10
        escreva("Adivinhe o número de 1 a 10");
        escreva("Digite 0 para sair a qualquer momento");
        while (true) {  // Laço eterno
            int chute = leia("Número: ");
            if (chute == n) {
                escreva("Acertou! Era o " + n);
                break;
            }
            if (chute == 0)
                break;
        }
    }
    public static int leia(Object texto) {
        System.out.print(texto);
        return new java.util.Scanner(System.in).nextInt();
    }
    public static void escreva(Object texto) {
        System.out.println(texto);
    }
}
```

A função `random` retorna um número real de 0 a 1, ajustado para valores de 1 a 10. No caso de acerto ou de o usuário querer desistir, o comando '`break`' (quebrar laço ou interromper) termina o programa. Este comando existe na maioria das linguagens de programação, como C, C++, C#, Java, JavaScript, Python, Matlab, etc.

EXERCÍCIOS

1. Demonstre o uso de laços imprimindo os 50 primeiros múltiplos de 3.

2. Faça um programa em qualquer linguagem de programação para mostrar a sequência de Fibonacci, de acordo com o fluxograma apresentado na primeira seção deste capítulo.

3. Faça um programa para verificar se um número `n` entrado pelo teclado é primo. Isto pode ser feito dividindo-o sucessivamente o número entrado por valores `i`, onde `i` varia de 2 até `n-1`, e verificando o resto da divisão. Se `n%i` (resto da divisão de `n` por `i`) for zero para qualquer `i`, exiba a mensagem “Não é primo! ”, interrompendo a execução. Caso a condição anterior não ocorra, o número entrado é primo.

4. a) utilize laços para calcular o mínimo múltiplo comum de dois números entrados pelo usuário. b) faça uma função que receba dois números como parâmetros e retorne o MMC ao ponto de chamada.

5. a) utilize laços para calcular o máximo divisor comum de dois números quaisquer entrados pelo usuário. b) faça uma função que receba dois números como parâmetros e retorne o MDC ao ponto de chamada.

6. Escreva um programa que leia três dados: Investimento inicial (`I`), Taxa de juros (`J`) e número de meses (`N`), em seguida calcule e exiba uma tabela de juros compostos, com o valor total do investimento corrigido do mês zero até o mês selecionado. Dica: procure saber mais sobre “saída formatada”. Veja o exemplo de saída produzida usando o comando de impressão formatada `printf`:
`printf("%5d %,20.2f %,20.2f %,20.2f\n", n, Jn, Jt, I );`

```
Mês    juros no mês  juros total    Investimento
  0            0.00          0.00          100.00
  1            1.00          1.00          101.00
  2            1.51          2.51          102.51
  3            2.02          4.53          104.53
...
```



Capítulo 5
Vetores

5 VETORES

Introdução

Trabalhando com vetores

Acessando elementos de um vetor

Formas de percorrer um vetor

Modularização e vetores

Eficiência de algoritmos

Exercícios

INTRODUÇÃO



Um vetor em programação de computadores é formado por um conjunto de `n` variáveis de um mesmo tipo de dado, sendo `n` obrigatoriamente maior que zero. Essas variáveis (ou elementos) são identificadas e acessadas por um nome e um índice. Na maioria das linguagens de programação, o índice recebe valores de `0` (primeiro elemento) até `n−1` (último elemento), em outras, a contagem vai de `1` até `n`, como é o caso nas linguagens MatLab, SciLab, entre outras.

Considerando a analogia feita no Capítulo 1, é possível imaginar as variáveis de um vetor como um armário de gavetas, porém, em cada gaveta só é possível guardar um único dado. Além disso, todas as gavetas desse armário são do mesmo tamanho e aceitam apenas dados de um mesmo tipo.

Desse modo, as variáveis de um vetor são armazenadas em posições consecutivas de memória. No caso de um vetor de variáveis do tipo `byte`, para um vetor com `n` variáveis, a memória ocupada pelo vetor seria `n*1 = n` *bytes*. Para um vetor do tipo `int` (inteiro), onde para cada variável são necessários `4` *bytes*, a quantidade de *bytes* necessários para armazenar este vetor na memória RAM será de `n*4` *bytes*. Esta característica de posições consecutivas de memória permite, em algumas linguagens como Java, C e C++, fazer cópia de vetores de forma rápida, copiando-se blocos de memória.

Outra vantagem de usar vetores na manipulação de dados de um mesmo tipo é o uso de índices, que permite acesso direto a qualquer elemento do vetor. Por exemplo, é possível copiar trechos de um vetor sem ter que percorrer todos os elementos até chegar ao início do trecho a ser copiado, o que não é possível se usar estruturas mais sofisticadas como listas ligadas (construídas com ponteiros – acesso direto à memória RAM). O acesso por índice permite a linguagem determinar rapidamente a posição da memória ocupada por cada elemento do vetor, considerando que uma variável do tipo vetor aponta para o endereço de memória do primeiro elemento do vetor e, sendo conhecido o tamanho de cada elemento, basta multiplicar-se o índice pelo tamanho (em *bytes*) de cada elemento e somar a posição de memória inicial. Logo, o endereço (`&`) de memória do elemento `i` de um vetor é sempre a posição inicial mais tamanho de cada elemento vezes o índice:

```
&vetor[i] = &vetor + tamanho(tipo_do_dado) * i
```

TRABALHANDO COM VETORES

Em programação de computadores, quando um vetor é criado, é **instanciada**[15] uma variável do tipo vetor, sendo necessário se definir um nome, o tipo de dado e o tamanho do vetor. Na linguagem C, por exemplo, a definição do tamanho de um vetor ocorre antes

---

[15] Criar uma instância ou instanciar refere-se à criação de objetos de qualquer tipo.

de compilar o programa, ou seja, o programador deverá informar no código a quantidade de memória a ser reservada para o vetor, através de um número inteiro positivo ou constante que o contenha, especificando o número de elementos do mesmo tipo que a memória reservada irá comportar. Neste caso, uma variável não pode ser usada. Já na linguagem Java, é possível alocar memória em tempo de execução do programa. Por exemplo, é possível criar um vetor para armazenar as notas de uma turma de alunos, onde o número de alunos é uma variável a ser lida durante a execução do programa.

Em muitas linguagens, como Java e C, o processo de criar um vetor ocorre em dois passos distintos: primeiro se cria uma variável de referência para o vetor; em seguida se reserva a memória para um dado número de elementos do mesmo tipo. No exemplo da Figura 5.1 a criação da variável de referência de um vetor `v` define apenas a posição de memória em hexadecimal `0A`, onde será armazenado o seu primeiro elemento. O segundo passo da criação de um vetor é reservar (ou alocar) memória para todos os elementos do vetor. Assim, independente da linguagem de programação utilizada, o processo de **criação** ou **instanciação** de um vetor deve ocorrer através: 1) da criação da variável de referência; 2) alocação de memória para todos os elementos do vetor.

	Conteúdo do vetor v	Posição na RAM
		09
v[0]	-128	0A
v[1]	0	0B
v[2]	6	0C
v[3]	98	0D
v[4]	127	0E
v[5]	4	0F
		10

Figura 5.1. Exemplo de apresentação de um vetor com 6 posições, seus conteúdos e suas respectivas posições na RAM.

Note que a variável `v` acima agora se refere a uma **posição de memória** e não a um elemento de dado. Os elementos devem ser acessados através do **índice**, um por vez. Isso pode requerer uso de laços para leitura e impressão dos seus elementos, dependendo da linguagem de programação utilizada. O **índice** é sempre um **número inteiro**.

A seguir são apresentados exemplos de criação (procedimento para instanciar e alocar memória) de vetores em diferentes linguagens de programação.

Código 5.1. Pseudocódigo: alocando memória para um vetor e atribuindo valores.

```
Instanciar o vetor v1 de inteiro com tamanho 5 // ou
vetor v2 = vetor de inteiros com 100 elementos // ou ainda
vetor v3 = inteiro(10)
v1 = {4,1,10,2,3} // atribuindo valores a v1
```

Código 5.2. Java: alocando memória para um vetor e atribuindo valores.

```
import java.util.Scanner;
public class criar_vetor {
    public static void main(String []args){

        System.out.print ("Digite o numero de alunos:");
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();    // lê número de elementos
        float v1[] = new float[n]; // declara e aloca memória
        int   v2[] = {4,1,10,2,3}; // instancia com valores

        String nome[]; // apenas declara o vetor
        nome = {"Ana", "Maria", "Paula", "Raul", "Vitor" };
        System.out.println(nome[0]+" "+nome[2]);
        // exibe conteúdo de elementos 0 e 2 do vetor: Ana Paula
        System.out.println(nome);
        // exibe a representação interna do objeto (hashcode16)

    }
}
```

Código 5.3. Portugol Studio: alocando memória em tempo de execução.

```
programa {
    funcao inicio(){

        const inteiro tamanho = 0
        escreva("Digite o tamanho de vetor: ")
        leia(tamanho)
        inteiro vetor[tamanho] // declara e aloca memória
        //...
    }
}
```

Código 5.4. JavaScript: alocando memória para um vetor e atribuindo valores.

```
v1 = new Array();      // instância sem dimensionamento
v2 = new Array(5);     // instância com dimensionamento
v3 = new Array(4,1,10,2,3); // instância com valores iniciais
document.write(v3) // imprime o conteúdo dos elementos de v3
```

ACESSANDO OS ELEMENTOS DE UM VETOR

Após criar a variável de referência e alocar memória para todas as suas posições, o vetor está pronto para que se insira ou altere os dados de seus elementos. Veja na Figura 5.2 um exemplo de apresentação de um vetor, seus conteúdos e índices.

Conteúdo do vetor v	Índice ido vetor v	
-128	0	← i=0
0	1	
6	2	← i+2
98	3	
127	4	← i+4
4	5	

Figura 5.2. Exemplo de apresentação de um vetor de inteiros com 6 posições, seus conteúdos e seus respectivos índices.

No exemplo de vetor da Figura 5.2, definindo `n` como o tamanho do vetor, é possível usar a seguinte estrutura em pseudocódigo:

Código 5.5. Pseudocódigo: atribuindo valores à elementos de um vetor.

```
vetor v de inteiros com 6 posições (n=6) // declara e aloca v
// primeiro trecho de atribuições de valores
inteiro i = 0
v[i]   = -128
v[i+2] = 6
v[i+4] = 127
// segundo trecho de atribuições de valores
v[1] = 0
v[3] = 98
v[5] = 4
```

O primeiro trecho de instruções faz com que os elementos do vetor nas posições 0, 2 e 4 recebam valores -128, 6 e 127, respectivamente. Finalmente, é possível completar o preenchimento desse vetor usando o segundo trecho de instruções.

ATENÇÃO: Um erro comum em programação ocorre ao tentar inserir um dado em uma posição não existente do vetor, excedendo seu limite. Ocorre, por exemplo, quando tenta-se acessar uma posição maior que o tamanho do vetor, em memória não reservada para ele. Por exemplo, ao criar um vetor `v` com 6 posições em Java, é necessário atribuir valores em `v[0]`, `v[1]` até `v[5]`. Neste caso, é comum tentar acessar a posição 6, como com `v[6]=0`, gerando um erro de índice fora do limite. Esse tipo de erro é difícil de ser identificado enquanto se escreve o código, ou mesmo após a sua compilação, aparecendo o erro para o usuário somente em tempo de execução. No passado, quando os sistemas

operacionais não tinham muita segurança, um acesso indevido à memória travaria o computador. Hoje, na pior hipótese, apenas o programa travaria.

FORMAS DE PERCORRER UM VETOR

PERCORRER UM VETOR COM "PARA"

Como um vetor, após criado, tem tamanho fixo (sempre com número de elementos `n>0`), geralmente é indicado usar estruturas de repetição do tipo `para`, de forma a percorrer todos os elementos do vetor, assim tornando o código genérico para um vetor de qualquer tamanho `n`. Além disso, é natural percorrer (ou varrer) o vetor da posição `i=0` até a posição `i<n`, onde o índice `i` assume automaticamente os valores `0, 1, 2,` até `n-1`, em cada iteração do laço. No exemplo a seguir em pseudocódigo, um vetor `v` é criado com 6 posições e o laço `para` inicializa todos os seus elementos com o valor `0`.

Código 5.6. Pseudocódigo: percorrer um vetor com `para`.

```
Instanciar um vetor v de inteiro com 6 elementos (n=6)
para cada indice i, de i=0; até i<n; passo i=i+1 faça
    v[i] = 0
```

Código 5.7. Python: exemplo de lista[16] com `para`.

```
impar = [ 1, 3, 5, 7, 9]    # cria lista com valores
lista = []                  # cria ista vazia
for i in range(6):          # varia i de 0 até 5
     lista.append(0)        # adiciona valor 0 à lista
print impar[3]              # imprime 7
```

Código 5.8. Java: percorrer um vetor com `for`.

```
public class vetor_for{
    public static void main(String []args){
        int v[] = {4,1,10,2,3};    // instancia com valores

        // forma com valores
        for (int i=0; i<v.length ; i++) {  // v.length = tamanho
            System.out.print(v[i] + " ");
        }
        System.out.println();

        // forma alternativa
        for (int temp:v) {
            System.out.print(temp + " ");
        }
    }
}
```

[16] Em Python, "lista" é a estrutura mais simples de múltiplos elementos. Para usar *arrays* convencionais, é necessário importar bibliotecas como *Array* ou *Numpy.*

São demonstradas duas formas de se operar um vetor: através de valores de índice diretos (inicial e final) ou usando-se uma variável temporária (`temp`) que, neste caso, assume todos os valores contidos no vetor, um por vez, sequencialmente. O resultado é análogo para os dois casos.

Observe que a variável `n` no laço do exemplo anterior foi substituída pela propriedade `length` que, em Java, retorna o número de elementos do vetor. No caso, `v.length` retorna o valor 5. Em C e C++ é possível se calcular o número de elementos dividindo o total de memória ocupado pelo vetor pela memória ocupada por cada elemento dele, de forma que `n = sizeof(v)/sizeof(v[0])`. Em Python, a biblioteca `numpy` disponibiliza o método `len(v)` que retorna o número de elementos no vetor `v`.

A vantagem de se usar a estrutura `para` é permitir embutir na própria sintaxe da instrução: declaração e inicialização do índice, incremento e condição de saída do laço.

PERCORRER UM VETOR COM “ENQUANTO”

Se o programador preferir, ou o problema a ser resolvido exigir, o vetor pode alternativamente ser percorrido utilizando-se uma estrutura de repetição do tipo `enquanto`:

Código 5.9. Pseudocódigo: percorrer um vetor usando `enquanto`.

```
n=6
vetor v de inteiros com n elementos
inteiro i=0
enquanto i<n faça {
    v[i] = 0
    i=i+1
}
```

Código 5.10. Java: imprimir um vetor usando `while`.

```
public class vetor_while{
    public static void main(String []args){

        int i = 0;
        int v[] = {4,1,10,2,3}; // instancia com valores
        while (i < v.length) {
            System.out.print(v[i++] + " ");
        }
    }
}
```

O código usando a estrutura de repetição `enquanto` produz o mesmo resultado que usando com `para`, porém, usando mais instruções: a criação do contador `i` e o incremento `i=i+1`. No exemplo em Java, o incremento é feito com o comando `i++`.

PERCORRER UM VETOR COM “FAÇA-ENQUANTO”

Outra opção de laço é usar a estrutura `faça-enquanto`, ou `do-while` existente em muitas linguagens de programação.

Código 5.11. Pseudocódigo: percorrer um vetor usando o `faça-enquanto`.

```
n=6
vetor v de inteiro com n elementos
inteiro i=0
faça {
    v[i] = 0
    i=i+1
} enquanto i<n
```

Código 5.12. Java: imprimir um vetor usando `do-while`.

```
public class vetor_do_while{
    public static void main(String []args){

        int i = 0;
        int v[] = {4,1,10,2,3}; // instancia com valores
        do {
            System.out.print(v[i]+" ");
            i=i+1;
        } while (i<v.length);

    }
}
```

OUTRAS FORMAS DE PERCORRER UM VETOR

Dependendo do problema a ser tratado, existem várias formas de se varrer um vetor usando estruturas de repetição para acessar cada elemento. Por exemplo, é possível percorrer o vetor do último elemento para o primeiro:

Código 5.13. Pseudocódigo: percorrer um vetor usando `para` na ordem inversa.

```
vetor v de inteiro com 6 elementos
n=6
para cada indice i, de i=n-1; até i==0; passo i=i-1 faça
    v[i] = 0
```

Código 5.14. Java: percorrer um vetor usando `para` na ordem inversa.

```
public class vetor_inverso{
    public static void main(String []args){

        int v[] = {4,1,10,2,3};     // instancia com valores
        for (int i=v.length-1; i>=0; i--) { // percorre inverso
            System.out.print(v[i]+" ");
        }

    }
}
```

Também, é possível percorrer somente alguns elementos do vetor, como para acessar os elementos que estão nos índices pares ou ímpares diferenciadamente:

Código 5.15. Pseudocódigo: percorrer um vetor usando `para`, com passo 2.

```
vetor v de inteiro com 6 elementos
n=6
para cada indice i, de i=0; até i<n-1; passo i=i+2 faça {
    v[i] = 0
    v[i+1] = 1
}
```

Código 5.16. Java: percorrer um vetor usando `para`, com passo 2.

```
public class vetor_indice_par{
    public static void main(String []args){

        int v[] = {4,1,10,2,3};     // instancia com valores
        for (int i=0; i<v.length; i=i+2) { // percorre pares
            System.out.print(v[i]+" ");
        }

    }
}
```

UMA APLICAÇÃO SIMPLES PARA PERCORRER UM VETOR

No exemplo a seguir será calculada a média de um vetor que representa as notas de uma turma com `n` alunos, alocados em tempo de execução. Assim, neste problema, a entrada será o tamanho do vetor e as notas dos `n` alunos. As saídas serão a média e o número de alunos com notas maiores ou iguais a média. Para simplificar, o código apresentado a seguir não valida (verifica se são válidos) os dados de entrada.

Código 5.17. Pseudocódigo: aplicação simples usando vetor.

```
// Entrada:
// instâncias e atribuições
real media, somador = 0
inteiro contador = 0
// leitura do número de alunos = tamanho do vetor
inteiro n = leia("Digite o número de alunos:");
vetor v de reais com n elementos
// leitura das notas dos n alunos
para cada indice i, de i=0; até i<n; passo i=i+1 faça
    v[i] = leia("Digite a nota do aluno " + i + ":");

// Processamento: soma, média e contador
para cada indice i, de i=0; até i<n; passo i=i+1 faça
    somador = somador + v[i] // soma
media = somador / n   // média
para cada indice i, de i=0; até i<n; passo i=i+1 faça
    se ( v[i] >= media ) // conta alunos>=media
        contador = contador + 1

// Saída:
escreva("Média da turma=" + media) // Saída
escreva("Alunos acima da média: " + contador)
```

Código 5.18. Java: aplicação simples usando vetor.

```
public class vetor_aplicacao_simples{
    public static void main(String []args){
```

```
        // Entrada: instâncias e atribuições
        float media, somador = 0;
        int contador = 0;
        // leitura do número de alunos = tamanho do vetor
        System.out.print("Digite o numero de alunos:");
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        float v[] = new float[n]; // vetor v de reais
        // leitura das notas dos n alunos
        for (int i = 0; i < v.length; i++) {
            System.out.println("Digite nota aluno " + i + ":");
            v[i] = scan.nextFloat();
        }

        // Processamento: soma, média e contador
        for (int i = 0; i < v.length; i++) {
            somador = somador + v[i];    // soma
        }
        media = somador / n;             // média

        for (int i = 0; i < v.length; i++) {
            if (v[i] >= media) {         // conta alunos>=media
                contador = contador + 1;
            }
        }
        // Saída:
        System.out.println("Média da turma=" + media);
        System.out.println("Alunos >= média: " + contador);
    }
}
```

Em linguagens interpretadas, como MatLab, Python e SciLab, a manipulação de vetores é simplificada para operações vetoriais básicas. Em Python, por exemplo, com o uso da biblioteca `numpy`, é possível se somar dois vetores, `v1` e `v2`, de mesmo tipo e tamanho, armazenando o resultado no vetor `vr`, com apenas uma instrução: `vr = v1 + v2`. Logo não é necessário usar laços para realizar operações de vetores elemento a elemento.

Código 5.19. Python: somando dois vetores.

```
# bibliotaca para manipular vetores multidimensionais
import numpy as np

# cria dois vetores com valores
v1 = np.array([5,2,3,4])
v2 = np.array([2,2,2,2])

# primeira versão para somar vetores
vr1 = np.array([0,0,0,0])
for i in range(4):  # range(4) = varrer posições 0, 1, 2, 3
    vr1[i] = v1[i] + v2[i]
print(vr1)

# segunda versão para somar vetores
vr2 = [v1[i]+v2[i] for i in range(4)]
print(vr2)

# ou, de forma bem mais simples
vr3 = v1+v2
print(vr3)
```

UMA APLICAÇÃO PARA TESTAR OS LIMITES DE UM VETOR: "DILATAÇÃO"

Como descrito no início deste capítulo, o programador deve tomar cuidado ao acessar os elementos de um vetor para não ultrapassar os seus limites alocados previamente, ou seja, para um vetor de tamanho `n`, índices `i` com `i<0` ou `i>=n` não existem. O exemplo a seguir ilustra este problema. Considere o problema de "dilatar" um vetor. O objetivo é criar um vetor `v1` de inteiros com `n` posições e um outro vetor `v2` de inteiros também com `n` posições. Cada posição `i` de `v2` armazena o cálculo do máximo entre cada elemento `i` em `v1` e seus vizinhos: à esquerda `v1[i-1]`, do próprio elemento `v1[i]` e do seu vizinho à direita `v1[i+1]`. Veja uma ilustração da operação de dilatação na Figura 5.3, onde `v1` é o vetor de entrada e `v2` é o vetor de saída, contendo a dilatação de `v1`, seguido do código para resolver este problema proposto.

Conteúdo do vetor v1	Conteúdo do vetor v2
-128	0
0	6
6	98
98	127
127	127
4	127

Figura 5.3. Exemplo de aplicação da função de "dilatação" de vetor, onde `v1` é o vetor de entrada e `v2` é o vetor com o resultado da "dilatação".

Código 5.20. Pseudocódigo: programa para “dilatar” um vetor.

```
// Inicializações
inteiro max=0
inteiro n = leia("Digite o tamanho do vetor:");
vetores v1 e v2 de inteiros com n elementos

// Entrada
para cada indice i, de i=0; até i<n; passo i=i+1 faça
    v1[i] = leia("Digite um número inteiro:");

// Processamento
para cada indice i, de i=0; até i<n; passo i=i+1 faça
    max = v[i]
    se i-1 >= 0 e max < v1[i-1] faça
        max = v1[i-1]
    se i+1 < n e max < v1[i+1] faça
        max = v1[i+1]
    v2[i] = max

// Saída
para cada indice i, de i=0; até i<n; passo i=i+1 faça
    escreva(" " + v2[i]);
```

Código 5.21. Java: programa para “dilatar” um vetor.

```
public class vetor_aplicacao_dilatar{
    public static void main(String []args){
        // Entrada:
        // instâncias e atribuições
        int max = 0;
        // leitura do tamanho do vetor
        System.out.print("Digite o tamanho do vetor:");
        Scanner scan = new Scanner(System.in);
        int n = scan.nextInt();
        int v1[] = new int[n]; // vetor v1 de inteiros
        int v2[] = new int[n]; // vetor v2 de inteiros
        // leitura dos n elmentos do vetor v1
        for (int i = 0; i < v1.length; i++) {
            System.out.print("Digite um inteiro  " + i + ":");
            v1[i] = scan.nextInt();
        }

        // Processamento:
        for (int i = 0; i < v1.length; i++) {
            max = v1[i];
            if (i-1>=0 && max < v1[i-1]) { // a esquerda
                max = v1[i-1];
            }
            if (i+1<n && max < v1[i+1]) { // a direita
                max = v1[1+i];
            }
            v2[i] = max;
        }

        // Saída:
        for (int i = 0; i < v1.length; i++) {
            System.out.println(" " + v2[i]);
        }
    }
}
```

No trecho de código anterior referente ao processamento, onde serão feitos os cálculos dos máximos, foi necessário verificar se o índice `i-1>=0` e `i+1<n` para não ultrapassar os limites do vetor `v1`. Na instrução `se`, convencionamos que a segunda condição lógica, `max < v1[i-1]`, é verificada somente se a primeira, `i-1 >= 0`, for verdadeira.

MODULARIZAÇÃO E VETORES

Como uma boa prática de programação, além de comentar os códigos e organizar com tabulação, como apresentado nos exemplos deste livro, é recomendável modularizar o código usando métodos, como apresentado no Capítulo 2. Todo sistema computadorizado de informação possui três partes bem definidas, agrupados em módulos de entrada, de processamento e de saída. Ao manipular informações armazenadas em vetores, é natural criar também pelo menos três módulos ou métodos[17], `lerVetor`, `processarVetor` e `imprimirVetor`, satisfazendo essa definição de sistema de informação.

MÉTODO DE ENTRADA

No último exemplo da seção anterior, no problema de "dilatar" um vetor, é possível criar um método de entrada chamado `lerVetor`, para ler o vetor, ou seja, inserir valores nas suas posições, como segue:

Código 5.22. Pseudocódigo: método para ler um vetor de inteiro tamanho n.

```
método lerVetor(inteiro n): retorna vetor de inteiro v[]
    vetor v de inteiros com n elementos
    para cada indice i, de i=0; até i<n; passo i=i+1 faça
        v[i] = leia("Entre com o elemento " + i + ":");
    retorne v
```

Código 5.23. Java: método para ler um vetor de inteiro.

```
    static int[] lerVetor(int n) {
        // leitura dos n elmentos do vetor v1
        int[] v = new int[n];
        for (int i = 0; i < n; i++) {
            System.out.print("Digite inteiro  " + i + ": ");
            v[i] = new java.util.Scanner(System.in).nextInt();
        }
        return v;
    }
```

[17] Os métodos podem ser chamados também de módulos, funções, subprogramas ou procedimentos. Existe uma convenção: quando um método tem argumento(s) (parâmetros) e um retorno é chamado de função, caso contrário, é chamado procedimento. Porém, poucas linguagens fazem distinção na sintaxe entre função procedimento, como a Pascal, tornando confusa esta convenção. Neste livro será usado método, para se aproximar da notação utilizada na UML (introduzida no Capítulo 7).

Esta versão de `lerVetor` em Java tem como argumento um inteiro `n`. Este método também tem uma variável de retorno, o vetor `v` alocado dentro do método, que é referenciado no programa principal. O programador deve dar atenção especial a assinatura dos métodos para não tentar modificar vetores que ainda não foram alocados na memória.

Assim, é importante ter atenção ao escopo do método. Neste exemplo, estamos lendo um vetor, referenciado no programa principal, por exemplo. Em linguagens de programação onde é necessário liberar espaço de memória alocada, como em C, poderia ser interessante convencionar que a alocação de memória fique fora do método, pois esta importante atividade pode passar desapercebido pelo programador e esquecer de liberar memória no final do programa. Então, uma boa prática de programação poderia usar a seguinte assinatura:

```
método lerVetor(inteiro v[], inteiro n): retorna vetor de inteiro
```

O vetor `v[]` foi alocado anteriormente no módulo principal, por exemplo.

MÉTODO DE PROCESSAMENTO

O segundo método para satisfazer um sistema de informação realiza o processamento, ou seja, os cálculos necessários para resolver o problema proposto. Nesse exemplo, é possível usar o método `dilataVetor`, tendo como entrada dois vetores `v1` e `v2`, e como saída o mesmo vetor `v2` contendo os dados de `v1` alterados ("dilatados").Seguem alguns exemplos de métodos, em algumas linguagens, para realizar o processamento da "dilatação" de um vetor:

Código 5.24. Pseudocódigo: método para "dilatar" um vetor.

```
método inteiro v2 = dilataVetor(inteiro v[], inteiro n):
    vetor v2 de inteiros com n elementos

    para cada indice i, de i=0; até i<n; passo i=i+1 faça
        max = v[i];
        se i-1>=0 e max<v1[i-1] faça
            max = v1[i-1]
        se i+1<n e max<v1[i+1] faça
            v2[i] = v1[i+1]
        v2[i] = max
    retorne v2
```

Código 5.25. Java: método para "dilatar" um vetor.

```
static int[] dilataVetor(int v[]) {
    int[] d = new int[v.length];
    for (int i = 0; i < v.length; i++) {
        int max = v[i];
        if (i-1 >= 0 && max < v[i - 1]) {       // a esquerda
            max = v[i - 1];
        }
        if (i+1 < v.length && max < v[i + 1]) {// a direita
            max = v[1 + i];
        }
        d[i] = max;
    }
    return d;
}
```

Código 5.26. Python: método para "dilatar" um vetor.

```
def dilataVetor(v):
    d = numpy.zeros(len(v))
    for i in range(len(v)):
        max = v[i]
        if i-1 >= 0 and max < v[i-1]:
            max = v[i-1]
        if i+1 < len(v) and max < v[i+1]:
            max = v[i+1]
        d[i] = max
    return d

# para testar
import numpy
v=range(5)
print dilataVetor(v)
```

MÉTODO DE SAÍDA

O terceiro método do nosso sistema de informação seria para apresentar os dados calculados durante o processamento. Veja um exemplo de saída de dados no método `imprimirVetor`:

Código 5.27. Pseudocódigo: método para imprimir um vetor.

```
função imprimeVetor(inteiro v[], inteiro n):
    para cada indice i, de i=0; até i<n; passo i=i+1 faça
        escreva(" " + v[i]);
```

Código 5.28. Java: método para imprimir um vetor de inteiro.

```
static void imprimirVetor(int v[]) {
    for (int i=0; i<v.length; i++) {
        System.out.print(" " + v[i]);
    }
    System.out.println();  // pula linha
}
```

Após a codificação desses três métodos, é possível reescrever o código "dilata" vetor apresentado na seção anterior, como segue:

Código 5.29. Pseudocódigo; aplicação para "dilatar" um vetor usando módulos.

```
// Instâncias e Atribuições
inteiro n = leia("Digite o tamanho do vetor:");

// Entrada
inteiro v1[] = lerVetor(n)

// Processamento
inteiro v2[] = dilataVetor(v1,n)

// Saída
imprimeVetor(v2, n)
```

Além do código ficar mais organizado, é possível reaproveitar os métodos **lerVetor** e **imprimirVetor** em outros programas que necessitem destes módulos.

Assim, sempre que possível é indicado usar métodos (funções ou procedimentos) para resolver os problemas de programação. Desta forma, o processo de construção de código é mais fácil, mais organizado, requer menos manutenção e produz menos erros e de mais fácil localização, graças à boa prática de modularização e reutilização de código. Verifique estas características no exemplo completo em Java:

Código 5.30. Java: programa "dilatar" um vetor, com métodos

```
public class Aplicacao_dilatar_vetor_com_metodos {

    public static void main(String[] args) {

        // Entrada: leitura do tamanho do vetor e valores
        int n = leia("Digite o tamanho do vetor: ");

        int v1[], v2[]; // vetores de inteiros de referência

        v1 = lerVetor(n); // Método de entrada de vetor

        // Método de processamento
        v2 = dilataVetor(v1);

        // Método de saída:
        imprimeVetor(v1);
        imprimeVetor(v2);
    }
```

```
    static int[] lerVetor(int n) {
        // leitura dos n elmentos do vetor v1
        int[] v = new int[n];
        for (int i = 0; i < v.length; i++) {
            v[i] = leia("Entre valor inteiro [" + i + "]: ");
        }
        return v;
    }
    static int[] dilataVetor(int v[]) {
        int[] d = new int[v.length];
        for (int i = 0; i < v.length; i++) {
            int max = v[i];
            if (i-1 >= 0 && max < v[i - 1]) {      // a esquerda
                max = v[i - 1];
            }
            if (i+1 < v.length && max < v[i + 1]) {// a direita
                max = v[1 + i];
            }
            d[i] = max;
        }
        return d;
    }
    static void imprimeVetor(int v[]) {
        for (int i = 0; i < v.length; i++) {
            escreva(" " + v[i]);
        }
        escreva("\n");  // pula linha
    }
    // Funções de ajuda: leia (int) e escreva (helpers)
    static int leia(Object texto) {
        escreva(texto);
        return new java.util.Scanner(System.in).nextInt();
    }
    static void escreva(Object o) {
        System.out.print(o);
    }
}
```

EFICIÊNCIA DE ALGORITMOS

A eficiência de um código (ou melhor, de um algoritmo) se mede analisando (ou contando) o número de instruções executadas. Este tipo de **análise de complexidade** é útil quando temos uma grande quantidade de dados a serem processados. Assim, um algoritmo mais eficiente (com menos passos) resolve um mesmo problema mais rápido. Imagine, por exemplo, que uma simulação física em tempo real é bem melhor que uma que leva várias horas para gerar resultados. Apesar de ser desejável medir a eficiência de um algoritmo considerando a medida de tempo, esta medida depende do sistema utilizado (hardware). Para resolver isto, a análise da complexidade de algoritmo se mede através da contagem do número de instruções executadas.

Para simplificar a contagem, podemos considerar apenas algumas instruções, como a quantidade de algumas condições lógicas. Por exemplo, considerando um algoritmo para

buscar um elemento `x` em um vetor de inteiros com `n` elementos, temos no **melhor caso**, apenas uma instrução, considerando que o elemento `x` está na primeira posição do vetor. Neste caso dissemos que o algoritmo de busca possui uma **complexidade assintótica**[18] **de melhor caso** $\Omega(1)$, ou **complexidade constante** (notação Bachmann–Landau, assintótica ou como é mais conhecida, *Big-O notation*). Verifique o teste de mesa a seguir considerando um vetor `v` com 5 elementos, contendo valores de 1 a 5, considere também que `x=1`.

Tabela 5-1. Teste de mesa para buscar um elemento em um vetor.

	Função Busca(int vet, int n, int x)	i	v[i]	Busca(v,5,1)
1	para i=0; i<n; i++	0		
2	se v[i]==x		1	
3	retorne i			0
4	retorne -1			
	Valores finais →			0

Por outro lado, este algoritmo de busca tem no **pior caso** `n` instruções de comparação, ou seja, se `x=5`, a instrução na linha 2 será executada 5 vezes. Neste caso, dissemos que o algoritmo de busca tem **complexidade assintótica de pior caso** $O(n)$, ou **complexidade linear**.

Quando $\Omega(n)=O(n)$, então dizemos que é um **algoritmo é ótimo**, ou $\Theta(n)$ (da ordem de `n`). Por exemplo, considere o algoritmo `buscaMaior` que encontra o maior elemento de um vetor de tamanho `n`. Neste caso, o algoritmo `buscaMaior` tem complexidade assintótica $\Theta(n)$.

Tabela 5-2. Algoritmo de busca do maior elemento em um vetor, com o número de instruções realizadas.

	Função buscaMaior(int vet, int n)	Número de instruções
1	int maior = v[0]	1
2	para i=1; i<n; i++	n-1
3	se maior < v[i]	n-1
4	maior = v[i]	< n-1
5	retorne maior	1
	Complexidade →	$f(n) <= 3n-1 = \Theta(n)$

A contagem do número de instruções pode ser realizada de forma simplificada somando todas as instruções que aparecem na última coluna da tabela anterior, assim `f(n) <= 1 + n-1 + n-1 + n-1 + 1 = 2 + 3(n-1) = 3n -1 <= kn`, para algum `k>=3`. Ou simplesmente, $f(n) = O(n)$. Como para este algoritmo de busca do maior elemento, o

[18] A complexidade assintótica de algoritmo considera o seu comportamento com um valor grande de dados a serem processados (`n` grande) e é abordado em detalhes em literaturas mais avançadas de programação.

melhor caso também é linear, ou seja, $\Omega(n)$. Podemos afirmar então que este algoritmo de busca é ótimo, ou seja, $\Theta(n)$.

Algoritmos muito estudados em análise assintótica são os de ordenação, para deixar os valores de um vetor com n elementos em ordem crescente ou decrescente. Considere o seguinte algoritmo de ordenação, chamado *bubble sort*[19]:

Tabela 5-3. Algoritmo *Bubble Sort* para ordenar vetor, com o número de instruções realizadas.

	Função ordena(int vet, int n)	Número de instruções
1	para i=0; i<n; i++	n
2	para j=0; j<n; j++	n*n
3	se v[i] > v[j]	n*n
4	int aux = v[i]	< n*n
5	v[i] = v[j]	< n*n
6	v[j] = aux	< n*n
7	retorne v	1
	Complexidade →	$f(n) = n*n = O(n^2)$

Uma versão um pouco melhor deste algoritmo *bobble sort* é apresentado a seguir. As linhas 2 até 6 são executadas seguindo a soma de uma progressão aritmética: $f(n) = n+n-1+n-2+...+1 = n*(a_1+a_n)/2 = n*(1+n)/2 = n/2 + n*n/2 <= n^2/2 <= n^2$. Portanto, apesar desta versão ser um pouco melhor que a versão anterior, ainda tem **complexidade assintótica quadrática**, ou $O(n^2)$.

Existem vários algoritmos de ordenação com diferentes ordens de complexidade[20], porém é possível se provar matematicamente que o mais eficiente entre eles não poderá ser melhor que $\Omega(n*\log n)$.

Tabela 5-4. Algoritmo *Bubble Sort* para ordenar vetor, versão melhorada, com o número de instruções realizadas.

	Função ordena2(int vet, int n)	Número de instruções
1	para i=0; i<n-1; i++	n-1
2	para j=i; j<n; j++	n+n-1+n-2+...+1
3	se v[i] > v[j]	n+n-1+n-2+...+1
4	int aux = v[i]	< n+n-1+n-2+...+1
5	v[i] = v[j]	< n+n-1+n-2+...+1
6	v[j] = aux	< n+n-1+n-2+...+1
7	retorne v	1
	Complexidade →	$f(n) = n*n/2 = O(n^2)$

[19] Veja uma simulação do algoritmo *bubble sort* com lego: https://youtu.be/MtcrEhrt_K0.
[20] Veja a comparação de vários algoritmos de ordenação: https://youtu.be/ZZuD6iUe3Pc.

EXERCÍCIOS

1. Criar um vetor para armazenar as notas de uma turma com `n` alunos e imprimir a maior e a menor nota da turma.
2. Criar dois vetores de mesmo tamanho, um para armazenar as notas de uma turma com `n` alunos. O outro vetor deve ser de `string`, onde para a mesma posição de uma nota `i` deve aparecer a palavra "`abaixo`" se a nota for abaixo da nota média da turma, ou "`acima`", caso contrário. Imprima ambos os vetores lado a lado, um elemento por linha.
3. Criar um vetor de inteiros com `n` elementos. Achar o menor valor e mostrar, junto com quantas vezes ele ocorre no vetor.
4. Criar um vetor `v1` de inteiros com `n` elementos com valores de 0 até 9. Criar um outro vetor `v2`, onde os elementos recebem a quantidade de ocorrências dos valores de 0 a 10, armazenando nas posições 0 a 10 de `v2`.
5. Criar um vetor de inteiros com `n` elementos e uma função que receba um vetor e duas posições `i` e `j`, efetuando a troca dos valores das posições `i` e `j` no vetor. Verifique na função se `0<=i<n` e `0<=j<n` antes de realizar a troca.
6. Criar um vetor de inteiros com `n` elementos e ordenar os seus valores.
7. O que ocorre dentro de um laço `para` que tem como objetivo inicializar `n` posições de um vetor `v` com índice `i` variando de `0` até `n-1`, quando houver a instrução `i=i+2` dentro do laço? E se o laço contiver a instrução `i=n`?
8. Criar um vetor de entrada com `n` posições com valores inteiros positivos e como saída criar um outro vetor também com `n` posições, onde a cada posição `i` é atribuído a cálculo do mínimo do seu vizinho de `v1` à esquerda `i-1`, do próprio elemento `i` e do seu vizinho à direita `i+1`.
9. Criar um vetor com `n` posições com valores inteiros positivos e, como saída, criar um outro vetor também com `n` posições, onde em cada posição `i` é atribuído a cálculo dos mínimos dos seus vizinhos de `v1` à esquerda `i-2` e `i-1`, do próprio elemento `i` e dos seus vizinhos à direita `i+1` e `i+2`. Generalize este código para os `m` vizinhos à esquerde e à direita.
10. O MMC (Mínimo Múltiplo Comum) de dois ou mais números inteiros é o menor múltiplo inteiro positivo comum a todos eles. Fazer uma função chamada MMC que recebe um vetor de números inteiros e retorna o MMC de todos. Veja um exemplo abaixo para calcular o MMC de 12 e 15:

a	b	/	
12	15	2	
6	15	2	
3	15	3	
1	5	5	
1	1	60	MMC=2*2*3*5

Capítulo 6
Matrizes



6 MATRIZES E VETORES MULTI-DIMENSIONAIS

Introdução

Instanciando matrizes

Acessando elementos de uma matriz

Formas de percorrer uma matriz

Aplicações usando matrizes

Exercícios

INTRODUÇÃO

Existem situações onde é necessário estender a definição de vetor para mais de uma dimensão de dados. Uma forma muito utilizada de manipulação de dados é em tabelas. Como exemplo, para listar todos os alunos de uma turma e suas notas em uma disciplina, as linhas da tabela podem armazenar identificações dos alunos na primeira coluna e nas colunas seguintes podem armazenar as notas da prova1, prova2, projeto e uma última coluna para armazenar a nota final do aluno. Analogamente a um vetor, em uma matriz cada elemento possui apenas um dado. Além disso, na maioria das linguagens de programação, todos os elementos de uma matriz são de um mesmo tipo de dado.

Um outro exemplo de matrizes muito usado, especialmente com a popularização dos dispositivos móveis como celulares e *tablets*, comumente trazendo câmeras digitais acopladas, ocorre no armazenamento e processamento de imagens digitais. Mas antes das câmeras digitais, já haviam *scanners* e *digitalizadores* e uma imagem é sempre representada por matrizes em um computador. A primeira matriz de imagem foi enviada em 1921, quando uma imagem foi enviada de Nova York a Londres usando cabos submarinos antes utilizados em telégrafos. A imagem transmitida representava 5 tons de cinza, logo, foi suficiente usar três *bits* para armazenar cada elemento da imagem.



Para processar uma imagem com apenas duas cores, um *bit* por elemento é suficiente, de forma que o valor 0 indique preto e 1, branco, por exemplo. Em uma matriz do tipo *booleana*, é possível convencionar que 0 = *falso* e 1 = *verdadeiro*. A imagem ao lado é de um QRCode, um código de barras bidimensional, que ilustra uma aplicação onde é possível usar uma imagem binária (`boolean`) para se armazenar dados, que podem ser recuperados usando-se um aplicativo leitor de QRCode, disponível para *Smartphones* (teste nesta imagem).



Os tipos de imagens mais comuns são: em níveis de cinza e coloridas. Geralmente, as imagens em níveis de cinza são armazenadas em matrizes do tipo `byte`, onde cada elemento pode representar até 256 (2^8) níveis de cinza (do preto ao branco). Nas imagens coloridas, é comum ter três ou quatro matrizes, uma para cada cor primária vermelho, verde e azul (no padrão RGB, do inglês: *Red, Green, Blue*) e, em alguns formatos, uma matriz alfa, contendo informação sobre a transparência de cada pixel (RGBA). Cada cor primária usa uma matriz do tipo `byte`. As imagens em níveis de cinza são a aplicação mais comum de vetores bidimensionais e as imagens coloridas no sistema RGB são exemplos de vetores tridimensionais. A imagem ao lado, por exemplo, tem uma resolução original de 1080 linhas por 1280 colunas em 3 matrizes (RGB).

INSTANCINDO MATRIZES

Na linguagem MatLab, uma matriz é definida com elementos da posição `(1,1)` até a posição `(L, C)`, onde `L` representa o número de linhas e `C` representa o número de colunas de uma matriz. Na maioria das linhagens de programação, no entanto, o índice começa no zero, sendo os elementos de uma matriz armazenados da posição `(0,0)` até `(L-1,C-1)`. Nos exemplos a seguir são apresentados alguns exemplos de instanciação de matrizes em diferentes linguagens de programação.

Código 6.1. Pseudocódigo: alocando memória em matriz e atribuindo valores.

```
Instanciar matriz m1 de inteiro com tamanho 2x3

m1 = {{4,1,10},{2,3,5}} // atribuindo valores
```

Note que os elementos da matriz m1 agora são vetores, estes, por sua vez, contendo, cada um, elementos do tipo alocado.

Código 6.2. Java: alocando memória para matriz e atribuindo valores.

```
import java.util.Scanner;
public class matriz_criar{
    public static void main(String []args){

        Scanner scan = new Scanner(System.in);
        System.out.print ("Digite o numero de alunos:");
        int n = scan.nextInt();
        System.out.print ("Digite o numero de provas:");
        int p = scan.nextInt();
        float m1[][] = new float[n][p]; // alocar matriz
        int   m2[][] = {{4,1,10},{2,3,5}}; // criar com valores
        System.out.println(m2[0][0]+" "+m2[0][1] +" "+m2[0][2]);

    }
}
```

Para alocar mais dimensões, é necessário considerar todas as dimensões na criação da variável de referência, e também ao alocar a memória adequada ao vetor multidimensional. No exemplo de uma imagem RGB, com três dimensões, em uma imagem com `L` linhas e `C` colunas, é necessário criar uma matriz com três dimensões: `d1,d2,d3`. Exemplo: `byte imagemRGB[][][] = new int[L, C, 3]`.

Código 6.3. JavaScript: alocando memória para matriz e atribuindo valores.

```
<script type="text/javascript">
m1 = new Array();      // instância sem dimensionamento
m2 = new Array(2,3);   // instância com dimensionamento
m3 = new Array(new Array(4,1,10),new Array(2,3,5));
                       // instância com valores
document.write(m3)     // imprime o conteúdo de m3
</script>
```

ACESSANDO ELEMENTOS DE UMA MATRIZ

Uma matriz está pronta para se inserir elementos ou alterar seus dados após instanciada e alocada na memória, em todas as suas dimensões. Veja uma forma de visualizar uma matriz 2D na Figura 6.1.

Figura 6.1. Exemplo de apresentação de uma matriz `m`, com 3 linhas e 2 colunas.

A Figura 6.1 ilustra uma matriz com 3 linhas e 2 colunas. Para visualizar seus elementos podemos usar os índices `i` e `j`, com valores para as linhas `0<=i<3` e para as colunas `0<=j<2`. Analogamente ao vetor, mas com uma dimensão a mais, a matriz recebe valores para os seus elementos, conforme a seguinte estrutura em pseudocódigo:

Código 6.4. Pseudocódigo: atribuindo valores em uma matriz.

```
Instanciar uma matriz m com 3 linhas e 2 colunas
m[0,0] = 5
m[0,1] = 6
m[1,0] = 7
m[1,1] = 8
m[2,0] = 9
m[2,1] = 7
```

FORMAS DE SE PERCORRER UMA MATRIZ

Analogamente ao vetor, uma matriz, após criada, possui tamanho fixo. Geralmente, para cada dimensão da matriz, é recomendado usar uma estrutura de repetição `para`, com o objetivo de tornar o código genérico para matrizes de quaisquer dimensões. Além disso, é natural percorrer (ou varrer) a matriz da linha `i=0` até a linha `i<L`, onde `L` representa o número de linhas de uma matriz. Para o Código 6.4, `0<=i<3` e as colunas `j`, onde `0<=j<C`. No exemplo a seguir em pseudocódigo, uma matriz `m` é criada com 6 elementos, sendo três linhas e 2 colunas e os laços `para` inicializam todos seus elementos com o valor 0.

Código 6.5. Pseudocódigo: percorrer uma matriz.

```
Instanciar uma matriz m com 3 linhas e 2 colunas
inteiros L=3, C=2
Para cada i, de i=0; até i<L; passo i=i+1 faça
    Para cada j, de j=0; até j<C; passo j=j+1 faça
        m[i,j] = 0
```

Código 6.6. Java: percorrer uma matriz.

```java
import java.util.Scanner;
public class matriz_percorrer{
    public static void main(String []args){

        int m[][] = {{4, 1, 10}, {2, 3, 5}}; // cria matriz m

        // imprimir matriz (cada linha de m em uma linha)
        for (int i = 0; i < m.length; i++) { // linhas i
            for (int j = 0; j < m[0].length; j++) { // colunas j
                System.out.print(m[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Existem várias formas de percorrer (ou varrer) uma matriz usando uma estrutura de repetição. Por exemplo, é possível percorrer uma matriz do primeiro elemento `m[0,0]` (convencionando como canto superior esquerdo) até o último elemento `m[L-1,C-1]` (convencionando como canto inferior direito da matriz), linha por linha, ou coluna por coluna. Esse tipo de varredura é chamado *raster*. A varredura inversa é chamada *anti-raster*, do último elemento inferior direito, até o primeiro elemento superior esquerdo.

APLICAÇÕES USANDO MATRIZES

Analogamente ao que foi feito no capítulo anterior sobre vetores, onde alocamos os vetores em tempo de execução através de métodos, é possível usar modularização para melhorar a organização, manutenção e reaproveitamento de código. Aqui é apresentado um método **`lerMatriz`** genérico para entrada de dados, ou seja, inserir valores nos elementos alocados na memória para uma matriz.

Código 6.7. Pseudocódigo: método para ler uma matriz de tamanho L x C.

```
Função inteiro m[][] lerMatriz(inteiro L, inteiro C):
    Instanciar e alocar uma matriz m de Reais com L x C
    Para cada i, de i=0; até i<L; passo i=i+1 faça
        Para cada j, de j=0; até j<C; passo j=j+1 faça
            m[i,j] = leia("Digite um número inteiro:");
```

Código 6.8. Java: método para ler uma matriz de inteiro.

```
static int[][] lerMatriz(int L, int C) {
   int m[][] = new int[L][C];
   for (int i = 0; i < m.length; i++)         // linhas i
      for (int j = 0; j < m[0].length; j++)   // coluna j
         m[i][j] = new java.util.Scanner(System.in).nextInt();
         // lê elemento i,j
   return m;
}
```

Linguagens como Python, Octav e Matlab permitem exibir o conteúdo de uma matriz simplesmente se imprimindo a variável do tipo matriz. Para as demais linguagens, sem esta funcionalidade, é muito útil para um sistema de informação ter uma função para apresentar os valores de uma matriz de forma clara. Segue um exemplo para uma função **`imprimirMatriz`**:

Código 6.9. Pseudocódigo: função para imprimir uma matriz de tamanho L x C.

```
Função imprimirMatriz(inteiro m[][], inteiro L, inteiro C):
    Instanciar e alocar uma matriz m de Reais com L x C
    Para cada i, de i=0; até i<L; passo i=i+1 faça
        Para cada j, de j=0; até j<C; passo j=j+1 faça
            escreva(" ", m[i,j]);
        escreva("\n"); // pula linha
```

Código 6.10. Java: método para imprimir uma matriz de inteiros.

```
    static void imprimirMatriz(int m[][]) {
        // imprimir matriz (cada linha de m em uma linha)
        for (int i = 0; i < m.length; i++) {        // linha  i
            for (int j = 0; j < m[0].length; j++)   // coluna j
                System.out.printf("%4d", m[i][j]); // formatada
            System.out.println();                   // pula linha
        }
    }
```

Note que o código em Java acima usa o método de impressão formatada `printf`, onde cada coluna terá sempre a largura exata de 4 caracteres decimais inteiros (`%4d`). Para formatar números reais, utilizamos a forma `printf("%4.1f", var_real)`, supondo que queiramos o número real exibido com 4 caracteres por coluna, com uma casa decimal depois da vírgula. A seguir é apresentado um código completo para somar duas **matrizes de valores reais** usando métodos.

APLICAÇÃO: SOMAR 2 MATRIZES

O exemplo a seguir calcula elemento a elemento a soma de duas matrizes, `m1` e `m2`, ambas com `L` linhas e `C` colunas, colocando o resultado da matriz `m3`.

Código 6.11. Java: soma de duas matrizes de reais.

```
public class matriz_aplicacao_soma_real{

    public static void main(String[] args) {
        // Entrada: leitura do tamanho do matriz e dados
        int L = (int) leia("Digite o numero de linhas: ");
        int C = (int) leia("Digite o numero de colunas: ");

        /* Entrada: Define duas matrizes de valores reais e lê
           os dados na sequencia com o método lerMatriz(L,C) */
        float m1[][] = lerMatriz(L,C),
              m2[][] = lerMatriz(L,C);

        // Método de processamento: cria e calcula matriz m3
        float m3[][] = somaMatrizes(m1, m2);

        // Método de saída:
        imprimirMatriz(m1);
        imprimirMatriz(m2);
        imprimirMatriz(m3);
    }

    static float[][] lerMatriz(int L, int C) {
        float m[][] = new float[L][C];
        for (int i = 0; i < m.length; i++) {      // linha  i
            for (int j = 0; j < m[0].length; j++) // coluna j
                // lê elemento i,j
                m[i][j]=leia("Entre elemento ["+i+"]["+j+"]:");
        }
        return m;
    }

    static float[][] somaMatrizes(float m1[][], float m2[][]) {
        float m3[][] = new float[m1.length][m1[0].length];
        for (int i = 0; i < m1.length; i++)
            for (int j = 0; j < m1[0].length; j++)
                m3[i][j] = m1[i][j] + m2[i][j];
        return m3;
    }

    static void imprimirMatriz(float m[][]) {
        // imprimir matriz (cada linha de m em uma linha)
        for (int i = 0; i < m.length; i++) {        // linha  i
            for (int j = 0; j < m[0].length; j++)   // coluna j
                System.out.printf("%6.2f", m[i][j]);
                // colunas de 6 caracteres com 2 casas decimais
            System.out.println();  // pula linha (fim da linha)
        }
        System.out.println();      // pula linha (fim da matriz)
    }

    // Método leia para real
    public static float leia(Object texto) {
        System.out.print(texto);
        return new java.util.Scanner(System.in).nextFloat();
    }
}
```

APLICAÇÃO: MULTIPLICAR DUAS MATRIZES

Como no exemplo anterior, podemos criar um método para retornar o resultado da multiplicação de duas matrizes de números reais. No entanto, dado o algoritmo de multiplicação de matrizes, onde multiplicamos os elementos de cada linha da matriz `A` pelos elementos de cada coluna da matriz `B`, será necessário mais um laço aninhado para implementar a lógica de programação. Lembremos também que o produto só existirá se o número de colunas de `A` for igual ao número de linhas de `B`.

Código 6.12. Java: método para multiplicação de duas matrizes de reais.

```
// retorma a multiplicação da matriz A pela matriz B, chamada:
// float[][] C = multiplica(A, B);

static float[][] multiplica(float[][] A, float[][] B)

   // verifica se n. de colunas de A é igual o de linhas de B
   if (A[0].length != B.length) throw new RuntimeException(
                                "Dimensão ilegal!"); // erro

   float[][] C = new float[A.length][B[0].length];   // aloca C
   for (int i = 0; i < C.length; i++)
      for (int j = 0; j < C[0].length; j++)
         for (int k = 0; k < A[0].length; k++)
            C [i][j] += A[i][k] * B[k][j];
   return C;
}
```

Podemos somar diretamente os valores multiplicados aos elementos da matriz `C` porque no Java os **valores padrões** para novas variáveis reais é **zero**.

Exercício: Como ficaria um método para calcular o determinante de uma matriz real 3x3?

Após resolver o exercício, veja uma possível solução genérica para o cálculo do determinante, além de outros exemplos de algoritmos para operações de cálculo com matrizes, no exemplo do Código 7.4 “Java: TAD Matriz.”, do capítulo 7.

APLICAÇÃO: MÉDIA DAS PROVAS DE UMA TURMA

O exemplo a seguir calcula a média para cada linha de uma matriz, que representa as notas de uma turma com `L` alunos (número de linhas), alocados em tempo de execução. Considere também que a primeira coluna da matriz armazena todas as notas da primeira prova, a segunda coluna armazena todas as notas da segunda prova. Finalmente a terceira coluna armazena a média das duas provas.

Código 6.13. Pseudocódigo: cálculo das médias de uma turma.

```
// Entrada
Inteiro L = leia("Digite o número de alunos:");

Instanciar matriz m de Reais com L linhas e 3 colunas
```

```
Para cada indice i, de i=0; até i<L; passo i=i+1 faça
    m[i,0] = leia("Digite a 1a nota do aluno " + i + ":");
    m[i,L] = leia("Digite a 2a nota do aluno " + i + ":");

// Processamento
Para cada indice i, de i=0; até i<L; passo i=i+1 faça
    m[i,2] = (m[i,0] + m[i,1]) / 2

// Saída
Para cada indice i, de i=0; até i<L; passo i=i+1 faça
    escreva("Média do aluno " + i + "=" + m[i,2]);
```

Observe que neste exemplo em particular, a entrada de dados não atribui valores para todos os elementos da matriz, sendo que a última coluna da matriz é reservada para calcular a média de cada aluno.

Código 6.14. Java: cálculo das médias de uma turma.

```
import java.util.Scanner;
public class matriz_aplicacao_turma{
    public static void main(String []args){

        Scanner scan = new Scanner(System.in);
        System.out.print("Digite o numero de Alunos: ");
        int n = scan.nextInt();
        System.out.print("Digite o numero de Provas: ");
        int p = scan.nextInt();
        float m[][] = new float[n][p + 1]; // alocar matriz

        // Entrada: ler as notas
        System.out.println("Digite notas de cada Aluno");
        for (int i = 0; i < n; i++)       // aluno i
            for (int j = 0; j < p; j++) { // prova j
                System.out.print("Aluno "+i+" prova "+j+": ");
                m[i][j] = scan.nextFloat();
            }

        // Processamento: calcula a média de cada aluno
        for (int i = 0; i < n; i++) {    // aluno i
            float soma = 0;
            for (int j = 0; j < p; j++)  // prova j
                soma += m[i][j];
            m[i][p] = soma / p;   // calcula a média
        }

        // imprimir matriz (cada linha de m em uma linha)
        System.out.println("p1 \tp2 \tmedia");
        for (int i = 0; i < n; i++) {       // linhas i
            for (int j = 0; j < p+1; j++)  // colunas j
                System.out.printf("%7.2f", m[i][j]);
            System.out.println(); // pula linha
        }
    }
}
```

APLICAÇÃO: IMAGEM

Considere uma imagem armazenada em um arquivo de nome “ufabc.pgm” no formato PGM[21], no conteúdo do arquivo a seguir, no formato texto, a 1ª linha do arquivo identifica o tipo do arquivo (P2 ou PGM), a 2ª linha traz as dimensões da imagem (Colunas x Linhas) e a 3ª linha o valor máximo de cada pixel.

```
P2
30 7
9
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 3 0 0 3 0 0 4 4 4 4 0 0 5 5 5 5 0 0 7 7 7 7 0 0 9 9 9 9 0
0 3 0 0 3 0 0 4 0 0 0 0 0 5 0 0 5 0 0 7 0 0 7 0 0 9 0 0 0 0
0 3 0 0 3 0 0 4 4 4 0 0 0 5 5 5 5 0 0 7 7 7 0 0 0 9 0 0 0 0
0 3 0 0 3 0 0 4 0 0 0 0 0 5 0 0 5 0 0 7 0 0 7 0 0 9 0 0 0 0
0 3 3 3 3 0 0 4 0 0 0 0 0 5 0 0 5 0 0 7 7 7 7 0 0 9 9 9 9 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```



Figura 6.2. Imagem gerada visualizando os dados com o software GIMP.

Nos exemplos seguintes usaremos a imagem abaixo, no formato PGM, agora em níveis de cinza, com valores variando do 0 (preto) até o 255 (branco).



Figura 6.3. Imagem de teste “lena.pgm”[22] antes e depois da binarização.

Os seguintes programas fazem a leitura do arquivo PGM e transformam a imagem de tons de cinza (256 níveis) para binária (branco e preto), processo conhecido como binarização,

---

[21] O software GIMP pode visualizar os formatos de imagens PGM, instale gratuitamente de www.gimp.org.

[22] Imagem popular de domínio público, disponível para download no site do livro.

muito usado na segmentação de imagens. Neste processo, pixels acima de um determinado valor (limite ou *threshold*) recebem 1, caso contrário recebem 0. A nova imagem é salva em outro arquivo PGM. O resultado do processamento, aplicado na imagem “lena.pgm”, é apresentado na Figura 6.3, onde foi usado um limiar 100, ou seja, apenas os pixels com valor acima de 100 (máximo 255) recebem valor 1 (branco).

Código 6.15. Pseudocódigo: aplicação de matriz usando imagem.

```
// Inicialização e entrada de dados
string str_in = leia("Qual o arquivo de entrada? ");
// Lê o nome da imagem, ex: "ufabc.pgm" ou  "lena.pgm"
string str_out = leia("Qual o arquivo de saída? ");
// Lê o nome do arquivo de saída, a conter a imagem binária

inteiro limiar = leia("Qual o limiar de corte? ");
// para ufabc.pgm use 1, para lena.pgm use limiar 100

scan = lerArquivo(str_in);

string tipoPGM = scan.lerProximaLinha();
inteiro C = scan.lerProximoInteiro();
inteiro L = scan.lerProximoInteiro();
inteiro ValorMaximo = scan.lerProximoInteiro();

//Instanciar e alocar matriz imagem com dimensões L x C
inteiro img_in[][] = inteiro[L][C];

// leitura imagem
para cada i, de i=0; até i<L; passo i=i+1 faça
    para cada j, de j=0; até j<C; passo j=j+1 faça
        img_in [i,j] = scan.lerProximoInteiro();

scan.FechaArquivo();

// processamento: calcular imagem binária
booleano img_out[][] = booleano[L][C];
para cada i, de i=0; até i<L; passo i=i+1 faça
    para cada j, de j=0; até j<C; passo j=j+1 faça
        se img_in[i][j] >= limiar
            img_out [i,j] = 1;
        senao
            img_out [i,j] = 0;

// saída: grava imagem do disco
saida = salvaArquivo(str_out);
escreva(saida,tipoPGM+”\n”); // formato pgm
escreva (saida, C + ” ”+L + ”\n“); // linha x coluna
escreva (saida, "1\n");// valor máximo é 1

Para cada i, de i=0; até i<L; passo i=i+1 faça
    Para cada j, de j=0; até j<C; passo j=j+1 faça
        escreva (saida, img_out[i][j];
    escreva (saida, ”\n”);

saída.FechaArquivo();
```

Código 6.16. Java: aplicação de matriz usando imagem.

```
public class matriz_aplicacao_imagem {

    public static void main(String[] args) {

        // Inicialização e entrada de dados
        string str_in = leia("Qual o arquivo de entrada? ");
        // Lê o nome da imagem, ex: "ufabc.pgm" ou  "lena.pgm"
        string str_out = leia("Qual o arquivo de saída? ");
        // Lê o nome do arquivo de saída
        String valor = leia("Qual o limiar de corte? ");
        int limiar = Integer.parseInt(valor);

        FileInputStream file_in = new FileInputStream(str_in);
        /* OBS: Se estiver usando o NetBeans, os arquivos
           tem que estar na pasta do projeto. */
        Scanner scan = new Scanner(file_in);

        // ler tipo, número de colunas, de linhas e valor máximo
        String tipoPGM = scan.nextLine();
        int C = scan.nextInt();
        int L = scan.nextInt();

        // processamento e saída: gera imagem e grava em disco
        PrintWriter file_out;
        file_out = new PrintWriter(new FileWriter(str_out));

        file_out.println(tipoPGM+"\n"+C+" "+L+"\n"+1);
        for (int i = 0; i < L; i++) {
            for (int j = 0; j < C; j++) {
                int aux = scan.nextInt();
                if (aux>=limiar) {
                    file_out.print("1 ");
                } else {
                    file_out.print("0 ");
                }
            }
            file_out.println(); // pula linha
        }
        file_in.close();
        file_out.close();
    }

    // leia String
    public static String leia(Object texto) {
        System.out.print(texto);
        return new java.util.Scanner(System.in).nextLine();
    }
}
```

Código 6.17. Python: aplicação de matriz usando imagem.

```
# bibliotaca para manipular vetores multidimensionais
import numpy as np

str_in = 'ufabc.pgm'
str_out = 'ufabc_bw.pgm'

limiar = 1

infile = open(str_in,'r')
tipoPGM = next(infile)

[C,L] = next(infile).split()
C = int(C)
L = int(L)
maxval = int(next(infile))

img = np.zeros([L,C],dtype = np.uint8)

# ler imagem
for i in range(L): # para cada linha
  linha = next(infile).split()
  j = 0
  for p in linha: # para cada elemento da linha
    img[i,j] = int(p)
    j+=1

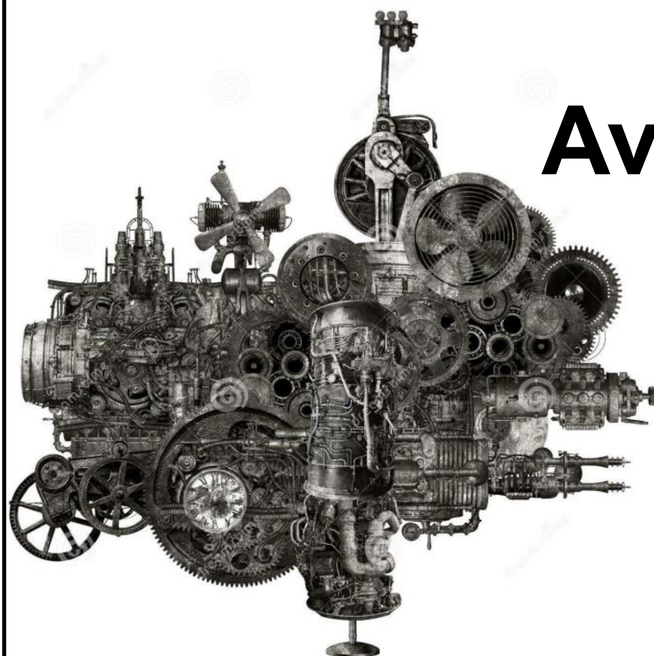
# cria imagem binária
imgBin = np.zeros([L,C], dtype = np.uint8)
imgBin[img >= limiar] = 1
# a instrução atribui 1 a todos os pixels em imgBin
# onde o valor em img for maior que limiar

# salva imagem binária no disco
oufile = open(str_out, 'w')
oufile.write(tipoPGM)
oufile.write(str(C)+ ' '+str(L)+'\n')
oufile.write(str(1)+'\n')
np.savetxt(oufile, imgBin , fmt='%1d')
oufile.close()
```

EXERCÍCIOS

1. Criar uma matriz para armazenar as `C` notas de uma turma e calcular a média de cada aluno, considerando uma turma com `L` alunos.
2. Em complemento ao exercício anterior, criar também um vetor de `String`, onde para cada posição `0<=i<L` seja armazenada a palavra "`reprovado`" se a nota for abaixo de 5, ou "`aprovado`", caso contrário.
3. Criar uma matriz para armazenar `L` linhas e `C` colunas. Calcular e exibir a soma dos elementos da diagonal principal e secundária da matriz.
4. Criar uma matriz para armazenar `L` linhas e `C` colunas. Calcular a soma dos elementos acima da diagonal da matriz.
5. Criar uma matriz `m1` para armazenar `L` linhas e `C` colunas. Criar uma outra matriz `m2` para armazenar `m1` transposta, com `C` linhas e `L` colunas, sendo que a primeira linha de `m2` contém os elementos da primeira coluna de `m1`. Isso se repete para as demais linhas de `m2` e colunas de `m1`.
6. Criar uma função/método para calcular e retornar o determinante de uma matriz 3 x 3 de valores reais recebida como parâmetro.
7. Criar um método que receba uma matriz qualquer de inteiros e imprima na tela o valor máximo e o valor mínimo encontrados (não retornando nada).
8. Criar um método que receba uma matriz de inteiros de qualquer dimensão e retorne um vetor contendo a somatória dos valores contidos nas colunas.
9. a). Criar um método que receba uma `matriz1` de inteiros positivos com `L` linhas e `C` colunas e retorne uma `matriz2`, de mesma dimensão, onde em cada posição `i,j` é atribuído o cálculo dos máximos entre o elemento da `matriz1` e seus quatro vizinhos. Os elementos são (`i-1,j`) (`i,j`), (`i+1,j`), (`i,j-1`) e (`i,j+1`).
   b). Generalize este código para os `m` vizinhos da `matriz1` à esquerde e à direita e os `n` vizinhos acima e abaixo.
10. Criar um método que receba duas matrizes de valores reais e retorne a multiplicação das duas matrizes.

Capítulo 7
Tópicos Avançados



7 TÓPICOS AVANÇADOS

Introdução

Paradigma Estruturado

Paradigma Orientado a Objetos

Tipo Abstrato de Dados

Introdução a Engenharia de Software

Processo e Gestão

Levantamento de Requisitos, Análise, Projeto e Testes

Ferramentas CASE

Exercícios

INTRODUÇÃO

Este capítulo introduz alguns conceitos usados em Programação Orientada a Objetos (POO) e em Engenharia de Software (ES) que serão úteis em estudos futuros. Não é o propósito deste capítulo cobrir completamente estes temas, mas sim proporcionar um ponto de partida para o estudo de linguagens orientadas a objetos, como Java, C++, Object-C e .NET, além de introduzir processos para desenvolver códigos envolvendo equipes usando princípios da Engenharia de Software.

PARADIGMA ESTRUTURADO

Antes da Programação Orientada a Objetos, os programas eram orientados a dados e a procedimentos, este estilo (ou paradigma) é chamado de **Programação Estruturada**. Neste paradigma estruturado, o programador define tabelas (ou registros), onde se podem armazenar variáveis de vários tipos de dados (diferentemente dos vetores e matrizes vistos neste livro, onde todos os dados devem ser de um mesmo tipo de dado). Por exemplo, é possível definir uma tabela para armazenar as informações de um aluno num contexto de um sistema acadêmico, chamada `Aluno`. Esta tabela `Aluno`, na verdade, pode ser considera um novo tipo de dados e é possível, por exemplo, "instanciar" uma variável (ou registro) `Julia` do tipo `Aluno`. Como atributos da tabela `Aluno`, é possível ter nome, matrícula, data de nascimento, ano de ingresso, etc. Observe que na tabela `Aluno` é importante também ter informação de curso e de disciplinas já cursadas. Estas informações podem ser "instâncias" de outras tabelas, como `Curso` e `Disciplina`, com seus respectivos atributos apropriados. Em Banco de Dados estas "instâncias" são possíveis através de um atributo representando a chave (estrangeira) de outra tabela. Além disso, cada tabela possui um atributo (chamado chave primária) que identifica o registro, por exemplo, `Julia`, que pode estar armazenada num registro de número 33.

Para poder visualizar melhor esta relação entre as tabelas `Aluno`, `Curso` e `Disciplina`, existe um modelo apropriado, chamado Diagrama Entidade Relacionamento (DER), como apresentado na Figura 7.1. Observe o relacionamento "`Aluno` faz `Disciplina`(s)". Se for considerar um contexto onde um aluno está cursando uma disciplina, existe a necessidade de incluir uma nova tabela, chamada, por exemplo, `Turma`. Assim, "`Turma` possui `Aluno`(s)" e "`Disciplina` possui `Turma`(s)".

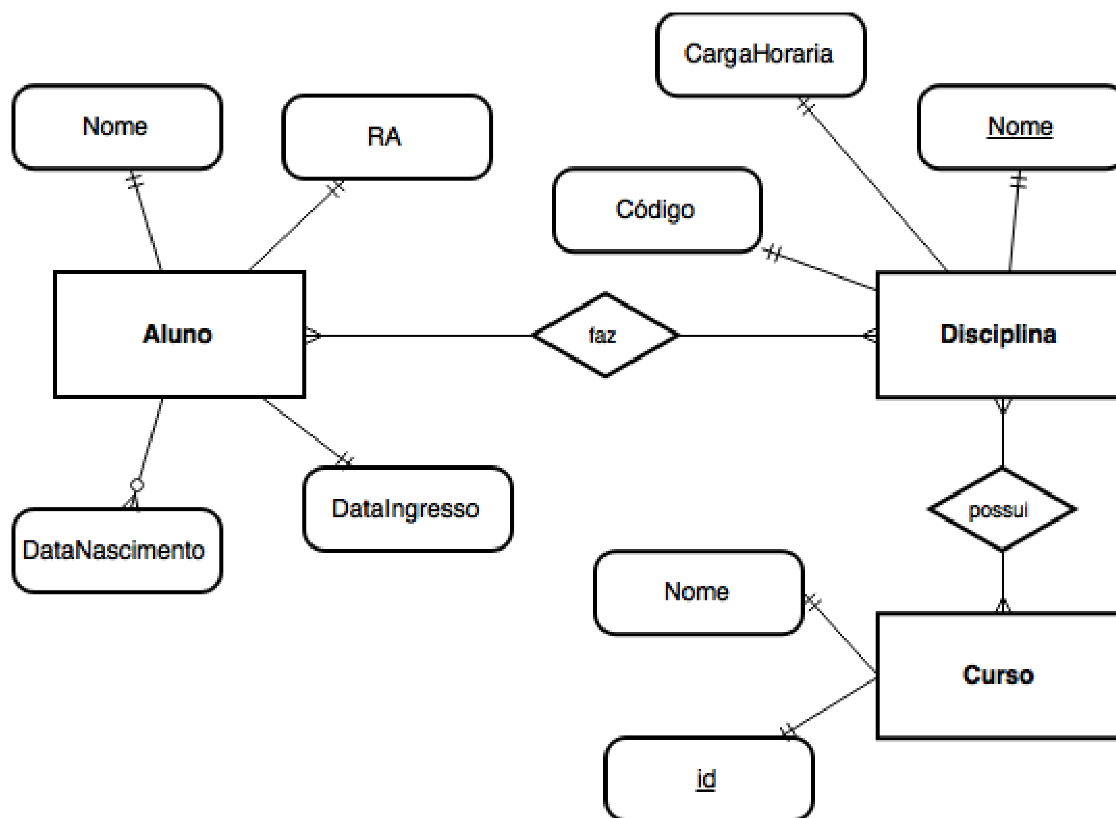


Figura 7.1. Exemplo de um Diagrama Entidade Relacionamento (DER).

Desse modo, para um sistema acadêmico completo, é necessário criar várias tabelas e seus relacionamentos. Todos esses dados das tabelas devem ser lidos (digitados) uma vez e armazenados em disco. Senão, toda vez que esse sistema acadêmico for executado, todos os dados deverão ser lidos novamente. Existem várias formas de guardar os dados de uma variável em disco: criar um arquivo texto, criar um arquivo binário ou criar tabelas em um Sistema de Gerenciamento de Banco de Dados (SGBD), como Oracle, JDBC, SQL, PostgreSQL, etc. Nas disciplinas de Banco de Dados, o aluno aprende como criar essas tabelas em um SGBD e também como usá-las dentro de uma linguagem de programação.

Esta forma de organizar vários tipos de dados em tabelas chama-se **encapsulamento de dados**. O grande desafio dos programadores e analistas de sistemas é definir corretamente essas tabelas e os seus relacionamentos. Além disso, os programadores devem criar funções e procedimentos de forma bem organizada, por exemplo, usando várias pastas. O paradigma estruturado não facilita esta organização. Isso já não ocorre no paradigma orientado a objetos.

Na programação estruturada não existe uma forma eficiente de organizar (encapsular) as funções específicas de uma tabela, como ocorre no encapsulamento dos dados, visto na seção anterior. Dessa forma, surgiu a necessidade de criar um novo paradigma de programação, que é a **Programação Orientada a Objetos** (POO), onde além de encapsular os dados, é possível encapsular as funções ou métodos que os processam. Assim, a tabela passou a se chamar **Classe**, um registro de uma tabela passou a ser chamado de instância de uma classe, também chamado de **objeto**. Como no exemplo anterior, para o sistema acadêmico, é possível criar a classe `Aluno` (que é um novo tipo de dados) e com ela instanciar uma variável `Julia` da classe `Aluno`.

Analogamente à programação estruturada, na POO existe uma forma de visualizar as classes com seus atributos e métodos e também os relacionamentos entre as classes. A Figura 7.2 apresenta os relacionamentos entre as classes, `Aluno`, `Curso` e `Disciplina`. Analogamente ao DER, existe a necessidade de criar também uma classe `Turma` entre o relacionamento "`Aluno` cursa `Disciplina`", para o caso de uma disciplina possuir várias turmas.

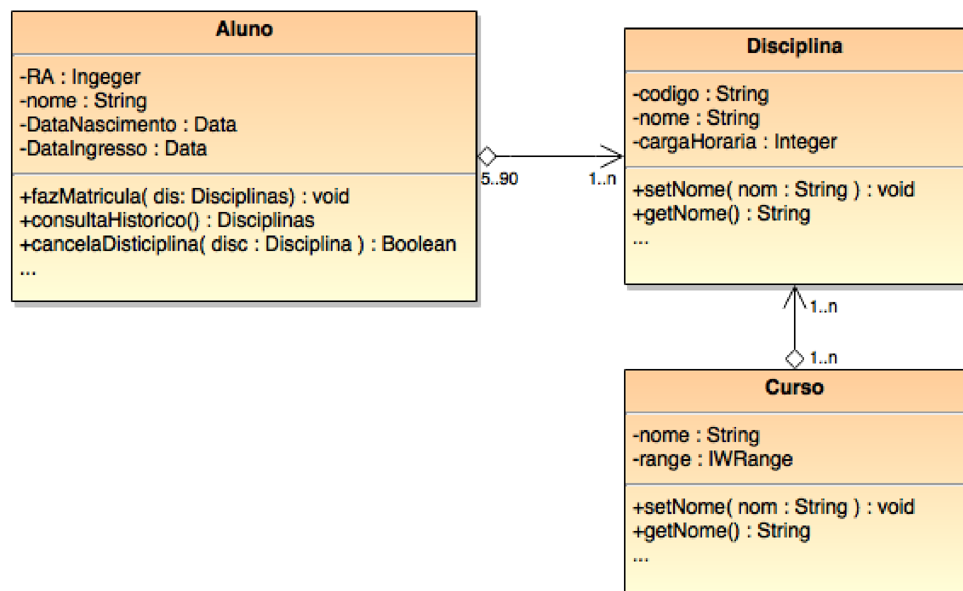


Figura 7.2. Exemplo de um Diagrama de Classes.

Além do encapsulamento de funções, alguns outros conceitos importantes na POO, são: herança, polimorfismo, associação, composição, agregação, padrões de projeto. Como ocorre na modelagem dos dados em um banco de dados, a modelagem das classes usando como exemplo um diagrama de classes é uma atividade que envolve grande concentração e abstração de um problema real a ser implementado. Todos esses conceitos e suas aplicações são detalhados em cursos avançados, como Programação

Orientada a Objetos e Engenharia de Software, fugindo do escopo deste livro, de caráter introdutório.

A seguir serão apresentados alguns exemplos de criação de classes e instanciação de objetos. Especificamente serão introduzidos os conceitos de Tipo Abstrato de Dados, construtor, sobrecarga de métodos e de operador, visibilidade e classes e métodos estáticos.

Para todos os programas apresentados neste livro, especificamente na linguagem de programação Java (que é orientada a objetos), foi necessário definir apenas uma classe para encapsular o método principal `main` (obrigatoriamente). Na parte de organização de código, incluímos mais alguns métodos para encapsular entrada, processamento e saída de dados. Apesar disto, não introduzimos formalmente conceitos de POO, logo, o leitor pode considerar que **todos os códigos apresentados neste livro, até o momento, seguem o paradigma estruturado**. No entanto, ao estudar POO o leitor verá que tudo o que foi apresentado neste livro será reaproveitado ao escrever códigos orientados a objetos (basta fazer corretamente o relacionamento entre as diferentes classes). Para começar, ao apresentar TAD na próxima seção, teremos que invocar os métodos de uma classe, ou melhor dizendo, os comportamentos dos objetos instanciados.

TIPOS ABSTRATOS DE DADOS

Um **Tipo Abstrato de Dados** (TAD) é um tipo de dados definido pelo programador, que pode vir a ser útil em múltiplos sistemas. Por exemplo, quando é definido um tipo vetor ou matriz, é criado um novo tipo de dados, além dos tipos usuais existentes nas linguagens de programação, como `int`, `float` e `string`. Quando se deseja somar dois números inteiros `a` e `b`, armazenando o resultando em `c`, é necessário usar o operador `+`, fazendo `c = a + b`. Ao usar-se um outro tipo, como `real` em vez de `inteiro`, o operador `+` é o mesmo, assim como `c = a + b`. Agora, ao usar o comando `+` para somar duas `Strings`, talvez a resposta não seja a esperada pelo programador: somar caractere por caractere, considerando os valores em ASCII, ou concatenar as duas `strings`, uma após a outra? No caso específico do tipo `String`, várias linguagens já oferecem bibliotecas para manipulação de dados, de forma que o programador não precisa implementar tudo do zero.

Ao criar uma classe, existe a opção de criar um método particular, que sempre será executado ao se instanciar um objeto desta classe. Esse método é chamado **construtor**. Em Java e C++, por exemplo, esse método construtor tem o mesmo nome da classe. Em Python o método construtor recebe o nome de `init()`. Em Delphi, se usa as palavras reservadas "*Constructor Create*".

Um recurso interessante existente na linguagem C++ é implementar um método com o nome `+`, esse recuso é chamado **sobrecarga de operador**. Alguns outros operadores que podem ser sobrecarregados em C++: +, -, /, %, ^, &, |, ~, !, <, <=, ++, +=, etc. Isto é útil por exemplo, quando queremos somar duas matrizes `A` e `B`, guardando o resultado na matriz

C. Em vez e implementarmos um método chamado somarMatrizes e fazer C = somarMatrizes(A,B), podemos escrever simplesmente, de forma mais elegante, C = A + B. Assim, estamos sobrecarregando o operador +.

Além da sobrecarga de operador, é possível fazer **sobrecarga de método**, ou seja, podem existir vários métodos com o mesmo nome, porém o que muda é a sua assinatura (argumentos de entrada). Assim, é possível implementar, por exemplo, um método construtor sem argumentos, ou com um argumento de um tipo (como inteiro), ou com um argumento de outro tipo (como real), ou com dois argumentos, etc. Todos esses métodos são distintos e serão executados conforme o número de argumentos e também conforme os seus tipos.

Outro conceito importante em POO é a **visibilidade** de classes, atributos e métodos. Por exemplo, pode ser útil em alguns sistemas definir os atributos login e senha de uma classe Pessoa com visibilidade privada, ou seja, somente os métodos da própria classe terão acesso a esses atributos. Podemos definir então um método chamado alteraSenha, com visibilidade pública, que poderá ser executado de qualquer lugar do sistema. A Tabela 7-1 traz as palavras reservadas usadas para definição da visibilidade de acesso em Java.

Tabela 7-1. Visibilidade de atributos, métodos e classes em Java.

Modificador	Classe (atual)	Pacote de classes	Subclasse	Mundo (todos)
Padrão (sem)	Sim	Sim	Não	Não
public	Sim	Sim	Sim	Sim
protected	Sim	Sim	Sim	Não
private	Sim	Não	Não	Não

Finalmente, um conceito de POO que também iremos usar neste capítulo é o de classes e métodos estáticos (static). Uma **classe estática** é criada quando não se deseja instanciar objetos. Por exemplo, é útil definir uma classe chamada ConverteTemperatura, de conversões de temperatura entre Kelvin, Fahrenheit e Celsius. Desta forma, não é necessário (e possível) instanciar um objeto de ConverteTemperatura, mas é possível usar um de seus métodos, por exemplo, temp=ConverteTemperatura.Kelvin2Celsius(300), onde temp irá receber o valor de 26.85.

EXEMPLO DE CLASSE: TAD FRAÇÃO

Supondo que queiramos representar números fracionários através de uma divisão de dois números inteiros, o **TAD fração** pode ser útil, tendo como atributos dois inteiros, por exemplo, numerador e denominador, e como funções de classe: simplifica, soma, subtração, divisão, multiplicação; dedicadas a tratar os dados deste tipo. A seguir são apresentadas as implementações iniciais de uma classe Fracao com alguns métodos, que são executados no programa principal.

Código 7.1. Pseudocódigo: TAD fração.

```
classe Fracao {
    inteiro num, den; // atributos da classe

    // métodos da classe

    Função Fracao (inteiro n, inteiro d) { // método construtor
        num = n;
        den = d;
    }
    Função Fracao soma(Fracao f) {
        retorna Fracao c(num*f.den+f.num*den,den*f.den);
    }
    Função escrevaFracao() {
        escreva("fração = "+num+"/"+den+"\n");
    }
}

principal {
    Fracao f1(2,3), f2(4,3); // cria frações f1 e f2
    Fracao fs = f1.soma(f2); // cria fração fs
    fs.escrevaFracao(); // imprime a fração resultante da soma
    // fs = (2*3+4*3)/3*3 = 18/9
}
```

Código 7.2. C++: TAD fração com sobrecarga de operador.

```
// tente executar este código em http://codepad.org
#include "stdio.h"
class Fracao {
  private:  // visibilidade privada
    int num, den; // atributos da classe

  public:  // visibilidade pública
    // métodos da classe
    Fracao(int n, int d){ // construtor
        num = n;
        den = d;
    }
    Fracao soma(Fracao f){ // função soma
        return Fracao(num*f.den + f.num*den, den*f.den);
    }
    // outra forma de implementar a soma sobrecarregando "+"
    Fracao operator+(Fracao f){
        return Fracao(num*f.den + f.num*den, den*f.den);
    }
    void escrevaFracao(){
        printf ("fração = %d/%d \n", num, den);
    }
};
// exemplos de uso dos métodos
int main() {
    Fracao f1(2,3), f2(4,3); // cria frações f1 e f2
    Fracao fs1 = f1.soma(f2); // cria fração fs1
    fs1.escrevaFracao(); /* imprime a fração resultante da soma
                            fs1 = (2*3+4*3)/3*3 = 18/9  */

    Fracao fs2 = f1 + f2; // cria fração fs2
    fs2.escrevaFracao();

    return 0;
}
```

Código 7.3. Java: TAD números complexos.

```
import javax.swing.JOptionPane;
public class Complexo {

    public double re = 0, im = 0;

    public Complexo() { } // Construtor padrão

    public Complexo(int re, int im) {
        // construtor com 2 2reais
        this.re = re;
        this.im = im;
    }
    public Complexo(Complexo c) {
        // Construtor de cópia
        this.re = c.re;
        this.im = c.im;
    }

//  OBS: O Java não oferece sobrecarga de operadores.
    public static Complexo soma(Complexo a, Complexo b) {
        Complexo r = new Complexo();
        r.re = a.re + b.re;
        r.im = a.im + b.im;
        return r;
    }
    public static Complexo subtrai(Complexo a, Complexo b) {
        Complexo r = new Complexo();
        r.re = a.re - b.re;
        r.im = a.im - b.im;
        return r;
    }
    public static Complexo multiplica(Complexo a, Complexo b) {
        Complexo r = new Complexo();
        r.re = a.re * b.re - a.im * b.im;
        r.im = a.re * b.im + a.im * b.re;
        return r;
    }
    @Override
    public String toString() { // converte para String
        String s = " " + re + " + " + im + "i ";
        return s;
    }

    public static void mensagem(Object o, int i) {
        JOptionPane.showMessageDialog(null, o, null, i);
    }

    public static void mensagem(Object o) {
        mensagem(o, -1);
    }
    void leia(Object o, double real, double imag) {
        re = leia(o.toString() + " Parte real.", real);
        im = leia(o.toString() + " Parte real.", imag);
    }
```

```
    static float leia(Object o, double val) {
        float r = Float.NaN;
        while (Float.isNaN(r)) {
            try {
                r = Float.parseFloat(
                    JOptionPane.showInputDialog(o, val));
            } catch (NumberFormatException e) {
                mensagem("Entrada inválida!", 2);
            }
        }
        return r;
    }

    // método de teste da classe
    public static void main(String[] args) {
        Complexo a = new Complexo(),
                 b = new Complexo(), c;
        a.leia("Entre complexo a.", 0.0, 0.0);
        b.im = a.re;
        b.re = a.im;   // b é ortogonal a 'a'
        c = soma(a, b);             // retorna a soma
        a = multiplica(b, c);      // retorna a multiplicação
        b = subtrai(a, c);        // retorna a subtração
        mensagem("Valores finais:\na = " + a
                + "\nb = " + b
                + "\nc = " + c); /* exibe a, b, e c como String,
                                    a conversão é automática
                                    com o método toString() */
    }
}
```

EXEMPLO CLASSE: TAD MATRIZ

Código 7.4. Java: TAD Matriz.

```
/*
 * Classe Matriz.java
 * Tipo final para trabalhar com objetos
 * do tipo Matriz de double de qualquer dimensão
 *
 */

final public class Matriz {
    private final int M;             // numero de linhas
    private final int N;             // numero de colunas
    private final double[][] data;   // Objeto do tipo Matriz

    // construtor de objetos: Matriz nome = new Matriz(M, N);
    public Matriz(int M, int N) {
        this.M = M;  // refere-se à variável global M
        this.N = N;  // refere-se à variável global N
        data = new double[M][N];
    }
    // Construtor de objeto matriz com vetor bi-dimensional
    public Matriz(double[][] data) {
        M = data.length;
        N = data[0].length;
        this.data = new double[M][N];
```

```
        for (int i = 0; i < M; i++)
            System.arraycopy(data[i], 0, this.data[i], 0, N);
    }

    // construtor de cópia
    public Matriz(Matriz A) { this(A.data); }

    // imprime matriz no formato padrão
    public void imprima(String o) {
        System.out.print(o); // usa o método toString()
        imprima();
    }
    public static void imprima(Matriz m) {
        m.imprima();
    }
    public void imprima() { // imprime o próprio objeto
        System.out.println(this); // usa o método toString()
    }

    @Override // Transforma em String
    public String toString() {
        String s = "\n[ ";
        for (int i = 0; i < M; i++) {
            for (int j = 0; j < N; j++)
                s += String.format(" %10.5f ", data[i][j]);
            if (i<M-1) s += ";\n  ";
        }
        s += " ]\n";
        return s;
    }
    public String toCSV() { // converte para formato CSV
        String s = "";
        for (int i = 0; i < M; i++) {
            for (int j = 0; j < N; j++)
                s += String.format(" %9.5f ;", data[i][j]);
            if (i<M-1) s += "\n";
        }
        s += "  \n";
        return s;
    }

    // Cria e lê nova matriz M x N
    public static Matriz leia(int M, int N) {
        Matriz A = new Matriz(M, N);
        return A.leia();
    }

    // Lê os elementos da matriz M existente com M.leia();
    public Matriz leia() {
        java.util.Scanner s = new java.util.Scanner(System.in);
        System.out.println("Lendo Matriz " + M + " x " + N);
        for (int i = 0; i < M; i++)
            for (int j = 0; j < N; j++) {
                System.out.print("Entre elemento da linha ["
                        + (i+1) + "] coluna [" + (j+1) + "]: ");
                this.data[i][j] = s.nextFloat();
            }
        return this;
    }
```

```java
    // cria e retorna matriz aleatória com valores de 0 até 1
    public static Matriz aleatoria(int M, int N) {
        Matriz A = new Matriz(M, N);
        for (int i = 0; i < M; i++)
            for (int j = 0; j < N; j++)
                A.data[i][j] = Math.random();
        return A;
    }

    // cria e retorna uma matriz identidade de N x N
    public static Matriz identidade(int N) {
        Matriz I = new Matriz(N, N);
        for (int i = 0; i < N; i++)
            I.data[i][i] = 1;
        return I;
    }

    // troca linhas i e j
    private void troca(int i, int j) {
        double[] temp = data[i];
        data[i] = data[j];
        data[j] = temp;
    }

    // cria e retorna a matriz transposta do objeto evocado
    // exemplo: Matriz B = A.transposta;
    public Matriz transposta() {
        Matriz A = new Matriz(N, M);
        for (int i = 0; i < M; i++)
            for (int j = 0; j < N; j++)
                A.data[j][i] = this.data[i][j];
        return A;
    }

    // retorma a soma das matrizes A e B
    // Matriz C = A.mais(B);
    public Matriz mais(Matriz B) {
        Matriz A = this;
        if (B.M != A.M || B.N != A.N)
            throw new RuntimeException("Dimensão ilegal.");
        Matriz C = new Matriz(M, N);
        for (int i = 0; i < M; i++)
            for (int j = 0; j < N; j++)
                C.data[i][j] = A.data[i][j] + B.data[i][j];
        return C;
    }

    // retorma a soma da matrizes A com uma matriz escalar
    // Matriz C = A.mais(valor);
    public Matriz mais(double e) {
        Matriz A = this;
        Matriz C = new Matriz(M, N);
        for (int i = 0; i < M; i++)
            for (int j = 0; j < N; j++)
                C.data[i][j] = A.data[i][j] + e;
        return C;
    }

    // retorma a subtração das matrizes A e B
    // Matriz C = A.menos(B);
    public Matriz menos(Matriz B) {
```

```
        Matriz A = this;
        if (B.M != A.M || B.N != A.N)
            throw new RuntimeException("Dimensão ilegal.");
        Matriz C = new Matriz(M, N);
        for (int i = 0; i < M; i++)
            for (int j = 0; j < N; j++)
                C.data[i][j] = A.data[i][j] - B.data[i][j];
        return C;
    }

    // retorma a subtração da matrize A e uma matriz escalar
    // Matriz C = A.menos(valor);
    public Matriz menos(double e) {
        Matriz A = this;
        Matriz C = new Matriz(M, N);
        for (int i = 0; i < M; i++)
            for (int j = 0; j < N; j++)
                C.data[i][j] = A.data[i][j] - e;
        return C;
    }

    // retorna booleano: é A == B? com A.igual(B)
    public boolean igual(Matriz B) {
        Matriz A = this;
        if (B.M != A.M || B.N != A.N)
            return false;
        for (int i = 0; i < M; i++)
            for (int j = 0; j < N; j++)
                if (A.data[i][j] != B.data[i][j]) return false;
        return true;
    }

    // retorma a multiplicação da matriz por um escalar
    // Matriz C = A.vezes(valor);
    public Matriz vezes(double d) {
        Matriz A = this;
        Matriz C = new Matriz(A.M, A.N);
        for (int i = 0; i < A.M; i++)
            for (int j = 0; j < A.N; j++)
                    C.data[i][j] += (A.data[i][j] * d);
        return C;
    }

    // retorma a multiplicação das matrizes A e B
    // Matriz C = A.vezes(B);
    public Matriz vezes(Matriz B) {
        Matriz A = this;
        if (A.N != B.M)
            throw new RuntimeException("Dimensão ilegal.");
        Matriz C = new Matriz(A.M, B.N);
        for (int i = 0; i < C.M; i++)
            for (int j = 0; j < C.N; j++)
                for (int k = 0; k < A.N; k++)
                    C.data[i][j] +=
                       (A.data[i][k] * B.data[k][j]);
        return C;
    }

    // retorma a multiplicação das matrizes A e B em C
    // Matriz C = vezes(A, B);
```

```
    public Matriz vezes(Matriz A, Matriz B) {
        if (A.N != B.M) throw new RuntimeException(
                              "Dimensão ilegal.");
        Matriz C = new Matriz(A.M, B.N);
        for (int i = 0; i < C.M; i++)
            for (int j = 0; j < C.N; j++)
                for (int k = 0; k < A.N; k++)
                    C.data[i][j] +=
                         (A.data[i][k] * B.data[k][j]);
        return C;
    }

    // resolve o determinante de uma matriz recursivamente:
    // double d = deterninante(M);
    public static double determinante(Matriz A) {
        return A.determinante();
    }

    // resolve o determinante do objeto atual:
    // double d = M.deterninante();
    public double determinante() {
        double det;
        if (M!=N) {
            System.out.println("Erro: Matriz não quadrada!");
            return Double.NaN;
        }
        if (N == 1)
            det = data[0][0];
        else if (N == 2)
            det = data[0][0] * data[1][1]
                - data[1][0] * data[0][1];
        else {
            det = 0;
            for (int j1 = 0; j1 < N; j1++) {
                double[][] m = new double[N - 1][];
                for (int k = 0; k < (N - 1); k++) {
                    m[k] = new double[N - 1];
                }
                for (int i = 1; i < N; i++) {
                    int j2 = 0;
                    for (int j = 0; j < N; j++) {
                        if (j == j1) {
                            continue;
                        }
                        m[i - 1][j2] = data[i][j];
                        j2++;
                    }
                }
                Matriz S = new Matriz(m);
                det += Math.pow(-1.0, 1.0 + j1 + 1.0)
                     * data[0][j1] * determinante(S);
            }
        }
        return det;
    }

    // classe principal para testes
    public static void main(String[] args) {
        double[][] m = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9} };
        Matriz X = new Matriz(m);
        imprima(X);
```

```
        System.out.println();
        Matriz teste = Matriz.leia(2, 2);
        teste.imprima("\nteste");
        Matriz A = Matriz.aleatoria(5, 5);

        A.troca(1, 2);
        A.imprima("A");

        Matriz B = A.transposta();
        B.imprima("B = A'");

        Matriz C = Matriz.identidade(5);
        C.imprima("C");

        A.mais(B).imprima("A x B");
        B.vezes(A).imprima("B x A");

        System.out.println(
        (A.vezes(B).igual(B.vezes(A))?"iguais":"diferentes"));
        System.out.println("Determinante de A = "
                           + A.determinante());
        Matriz b = Matriz.aleatoria(5, 1);
        b.imprima();
    }
}
```

OUTROS EXEMPLOS DE APLICAÇÕES DE CLASSES TAD

TAD polinômio, tendo como atributos um vetor com os coeficientes do polinômio armazenados em um atributo do tipo vetor. Como métodos de classe é possível implementar: soma, subtração, multiplicação, divisão, raízes, valor, etc.

TAD vetor, onde seria possível criar uma classe chamada `Vetor`, tendo como atributo, por exemplo, um vetor de inteiros. Como métodos para esta classe, é possível implementar valor máximo, valor mínimo, somar vetores, produto escalar, produto vetorial, atribuir valor em uma posição, retornar valor de uma posição, excluir uma posição, ordenar vetor, etc.

Outras variantes do TAD vetor poderia ser **TAD pilha** (a inserção e remoção de qualquer elemento deve ocorrer em apenas um extremo do vetor). Este tipo de vetor também é conhecido como vetor LIFO (*Last-In, First-Out*). Veja uma ilustração de um vetor para manipular uma pilha na Figura 7.3.

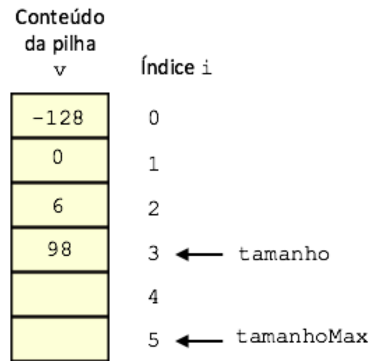


Figura 7.3. Ilustração de uma pilha.

Nesse tipo de manipulação de vetor como pilha é necessário ter cuidado para não acessar uma posição não alocada do vetor. Por exemplo, se for inserir um novo elemento em uma pilha, quando o `tamanho = tamanhoMax`, será necessário alocar memória para um outro vetor maior e fazer a cópia da fila no novo vetor e finalmente incluir o novo elemento. Uma solução para não ter que fazer cópia de vetor, seria usar a estrutura de ponteiros, porém isso não existe em algumas linguagens de programação como a Java.

Outro exemplo de TAD vetor pode ser o **TAD Fila**, funcionando como uma fila em um caixa de um banco, onde o primeiro elemento que entra na fila é o primeiro que sai. Esse tipo de vetor também é conhecido como vetor FIFO (*First In, First Out*). Veja uma ilustração de um vetor para manipular uma fila na Figura 7.4.

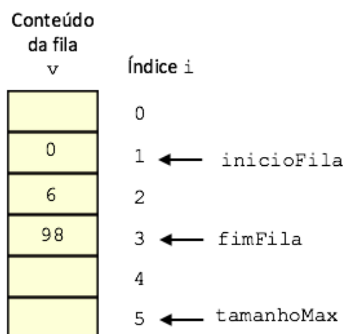


Figura 7.4. Ilustração de uma fila.

Observe que tanto TAD pilha, como TAD fila, herdam características de TAD vetor. Ou seja, TAD pilha e TAD fila "são tipos de" TAD vetor (subclasses). Quando é possível escrever "é tipo de" ao relacionar duas classes, geralmente indica uma herança. Assim, os atributos e métodos de uma classe mãe podem ser herdados pelas classes filhas, deste que a classe mãe definindo o escopo de acesso (visibilidade) dos seus atributos e métodos.

Mas qual é a vantagem de criar estas classes específicas de TAD? Quando bem executada a implementação da classe (bem feito e sem erros), a mesma poderá ser reutilizada em vários outros sistemas, poupando o trabalho de escrever novamente os mesmos algoritmos.

Agora serão apresentados alguns conceitos úteis para o desenvolvimento de soluções complexas baseadas em linguagens de programação.

Uma definição de **Software**: "*Qualquer programa ou grupo de programas que instrui o hardware sobre a maneira como ele deve executar uma tarefa, inclusive sistemas operacionais, processadores de texto e programas de aplicação"* [23].

Assim, um software é um programa de computador escrito para executar alguma tarefa e atender a alguma finalidade. Porém, existem diversas formas de construir um software de qualidade.

Uma analogia pode ser feita com a construção de um prédio: para que o prédio tenha o menor número de falhas, um pedreiro não pode iniciar a construção sem um planejamento envolvendo vários outros profissionais, como arquiteto, engenheiro civil, engenheiro elétrico, etc.

Infelizmente, muitos softwares são desenvolvidos de forma caótica, onde o programador tem uma vaga ideia do software a ser construído quando começa a programar. Isso produz um grande retrabalho, pois nem sempre o cliente sabe exatamente o que quer, ou não sabe passar as informações necessárias para o programador. Quando se trata de uma equipe envolvendo várias pessoas, então o problema será ainda maior. O sucesso desse tipo de software está diretamente relacionado com a experiência dos envolvidos, ou seja, se já construíram softwares semelhantes. Para não correr esse risco, existe uma área da ciência que estuda as boas práticas de construir software.

A **Engenharia de Software** é a área da ciência que estuda **processos**, **métodos/técnicas** e **ferramentas** que auxiliam as pessoas a construir software com boa qualidade, ou seja, correto, funcional, eficiente, reutilizável, com fácil manutenção, feito no prazo e no orçamento predefinido.

Um software pode ser genérico, como um sistema operacional, um editor de texto, ou mesmo uma ferramenta CASE (do inglês *Computer-Aided Software Engineering*). Um software também pode ser adquirido sob encomenda para uma aplicação específica, como a automatização de uma clínica médica. Existem diversas outras classificações de software, como customizado (exemplo ERP – do inglês *Enterprise Resource Planning*), legado, embarcado, Sistemas de Processamento de Transação, Sistemas de Informação Gerencial, Sistemas Especialistas, etc.

---

[23] Definição encontrada em michaelis.uol.com.br.

Como uma ilustração, considere uma esfera quase perfeita, com faces triangulares, como um software completo (observe que aqui é **redefinido** o conceito de **software**, onde além de incorporar o código, incluí também diversos outros documentos que facilitam o seu entendimento e o seu desenvolvimento). Em cada face da esfera existe uma visão do software, por exemplo, em uma face existe a descrição do sistema como a visão mais abstrata. Esta descrição pode ser detalhada em outras faces da esfera através de vários documentos ou **artefatos** de software, como exemplo, um documento que detalha um cenário do software. Cada artefato representa uma visão diferente do mesmo software. Um outro exemplo de artefato é o código necessário para implementar o sistema. Este código estaria em outra face da esfera, por exemplo, oposta ao contexto, e representa a visão do programador e de como o computador deverá executar as instruções para satisfazer os requisitos levantados com o cliente. Em uma outra face pode existir também um tipo de teste, como o teste unitário, necessário para testar cada componente do software.

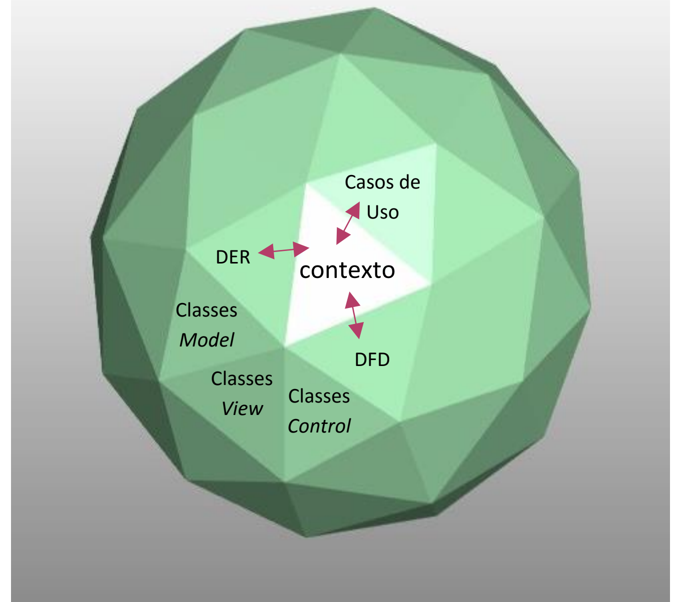


Figura 7.5. Conceito de artefatos de software como faces triangulares de uma esfera quase perfeita.

Indo além nesta visão da esfera, ilustrada na Figura 7.5, é possível imaginar que o cliente tem uma visão perpendicular ao contexto, ou seja, ele consegue ler e entender perfeitamente o que está escrito no contexto, porém, os artefatos mais distantes ao contexto na esfera são mais difíceis do cliente enxergar, consequentemente, de ler e entender o seu conteúdo. Do lado oposto ao contexto está o código, onde o cliente não tem visão alguma do seu conteúdo. Porém o computador, consegue interpretar e executar esta visão de código. Também, o computador consegue interpretar alguns outros artefatos, quando é usado uma ferramenta CASE sofisticada que transforma automaticamente diagramas em código (ou em outros tipos de diagramas), como será apresentado no final deste capítulo.

Outra informação útil nesta ilustração seria a ordem de “construir cada face” do sistema, ou seja, a precedências dos artefatos. Isso será apresentado a seguir.

As próximas seções deste capítulo apresentarão uma introdução aos conceitos de Engenharia de Software, usados na construção de software com qualidade. Detalhes desses conceitos são temas de estudos mais avançados.

PROCESSO E GESTÃO

Um **processo de software** é descrito como um conjunto de atividades bem definidas para construir software. Ou seja, o processo que organiza as atividades (agrupadas em **fases**) necessárias para construir um software. Por exemplo, em situações raras, é possível desenvolver um software seguindo as seguintes fases sequencialmente:

1) **Requisitos**: levantar os requisitos junto aos envolvidos;
2) **Análise**: desenhar vários diagramas abstratos (de auto nível de abstração) para identificar os dados, os cenários, as interfaces do software; entre outros;
3) **Projeto**: desenhar diagramas detalhados, qualquer informação de como o software será construído deve estar documentado nesta fase;
4) **Implementação** ou codificação: traduzir o projeto para uma linguagem de programação específica;
5) **Testes** e validação: seguido dos ajustes necessários para operação;
6) **Operação**: implantação no ambiente real de uso.

Esse fluxo **linear** (como um mnemônico: **RAPITO**) raramente ocorre, pois, por exemplo, geralmente é inviável levantar todos os requisitos de uma única vez. Mas é melhor do que desenvolver um sistema de forma caótica, sem processo algum. Um modelo mais natural para construir software seria percorrer essas 6 fases várias vezes, onde em cada iteração são acrescentas novas funcionalidades ao software. Este fluxo de processo é chamado **evolucionário**. Existe também o fluxo **iterativo**, onde uma ou mais fases são repetidas antes de se passar para a fase seguinte. No fluxo **paralelo**, algumas fases para desenvolver cada componente do software ocorrem concomitantemente e, no final, esses componentes são integrados. Esses fluxos classificam os vários modelos prescritivos clássicos existentes na literatura (**Cascata**, **RAD**, **Prototipagem**, **Espiral**, etc.).

O fluxo em que são realizadas estas 6 etapas, além do conjunto de artefatos produzidos, define **o ciclo de vida do desenvolvimento de software**.

Após a criação da metalinguagem UML (linguagem de modelagem unificada, em português) no início da década de 90, James Rumbaugh, Grandy Boock e Ivar Jacobson criaram um fluxo mais complexo de processo de software chamado **Processo Unificado**, seguindo um fluxo iterativo e incremental, dirigido à casos de uso e centrado na arquitetura.

Definido o processo adotado, o próximo passo é criar um cronograma (ou gráfico de Gantt) descrevendo todas as atividades que deverão ser realizadas, a ordem de precedência, a duração e os recursos necessários (como as pessoas) para cada atividade.

Um processo e um cronograma para o desenvolvimento de um software só podem ser elaborados após uma breve descrição do software, definindo assim um **contexto do sistema** (a ser exemplificado na próxima seção). Com essas informações, os recursos

deverão ser levantados, como pessoas, equipamentos, linguagens, bibliotecas, softwares (genéricos), custo e tempo para o desenvolvimento.

Outras informações também podem ser levantadas nesta fase inicial, como levantamento de riscos e seus planos de ações, definição de versões, atividades de controle de qualidade, como a **Revisão Técnica Formal** (RTF), estimativas usando **Análise por Ponto de Função**, etc. Todos os documentos gerados nestas fases de definição do processo e da gestão podem ser classificados como **artefatos de processo**.

O **CMMI** (*Capability Maturity Model Integration*) é uma forma de validar se uma empresa usa corretamente processos de software[24], ou seja, CMMI é um *metamodelo* de processo. Uma empresa pode ser classificada em 5 níveis de maturidade: 1: Inicial (Ad-hoc); 2: Gerenciado; 3: Definido; 4: Quantitativamente gerenciado; 5: Em otimização. Existe também um *metamodelo* específico para o Brasil, chamado MPS.BR (Melhorias de Processo de Software Brasileiro), mais acessível para pequenas e médias empresas de desenvolvimento de software[25]. Neste MPS.BR, os dois níveis iniciais do CMMI são divididos em mais dois níveis, resultando em 7 níveis de maturidade. Assim, fica mais fácil as empresas atingirem estes níveis iniciais de maturidade.

REQUISITOS

O levantamento de requisitos inicia com a descrição narrativa do software, ou descrição do **contexto** do sistema. Por exemplo, considere o seguinte contexto definido por um grupo de alunos da UFABC.

> *Um grupo de alunos deseja fazer um aplicativo no Android usando a linguagem de programação Java para ofertar carona na UFABC. Este sistema* ***caronaUFABC*** *irá criar uma lista de Pessoas, podendo ser Motoristas ou Passageiros. Os motoristas devem cadastrar uma lista de Carros, detalhando o tipo do carro, número de passageiros, etc. Este sistema deve criar uma entidade Carona, contendo como atributos origem e destino, dia e horário da carona e quanto o passageiro deve pagar para dividir as despesas do combustível, pedágio, etc. O sistema deve criar uma lista de passageiros, uma lista de motoristas com o(s) seu(s) carro(s), e uma lista de Carona, ligando todas as entidades necessárias. Existe também um módulo de gestão, onde o administrador consegue imprimir relatórios do uso do aplicativo, como Pessoas cadastradas, Caronas realizadas em períodos, etc. Tomar o cuidado de não permitir carona acima do número máximo de lugares do carro.*

[24] O CMMI é normatizado em cmmiinstitute.com
[25] O MPS.BR é normatizado em www.softex.br/mpsbr

Outras atividades podem ser realizadas no levantamento de requisitos, mas neste capítulo introdutório, vamos nos restringir apenas ao **contexto**, ou seja, à concepção básica do problema. Um modelo para esta fase pode ser o documento de visão do RUP[26], ou a norma IEEE 830. Além de textos explicando o que será desenvolvido, as prioridades, recursos necessários, é possível incluir alguns modelos de análise neste documento, como os apresentados a seguir.

ANÁLISE

Após o levantamento dos requisitos básicos, o próximo passo é fazer uma análise usando vários modelos, como os modelos funcionais: Diagrama de Casos de Uso, Diagramas de Fluxos de Dados, IDEF0 (do inglês *Integrated DEFinition Methods*) e o modelo de dados com o Diagrama Entidade Relacionamento. Esses modelos vão ajudar a validar os requisitos levantados junto com os envolvidos.

Esses modelos de análise definem uma nova linguagem de comunicação entre a equipe de desenvolvimento e o cliente. Dessa forma, o nível de abstração desses modelos deve ser apropriado para que o cliente consiga ler e acompanhar o que será desenvolvido. Por exemplo, detalhes de programação não devem aparecer nessa etapa, como fluxograma de métodos de classes e padrões de projeto.

Além do Diagrama de Casos de Uso, outros diagramas UML também podem ser incluídos desta etapa de análise: Diagrama de Atividades, Diagrama de Classes, Diagrama de Sequência e Diagrama de Estado.

PROJETO

Os modelos de análise apresentam como objetivo identificar o propósito, ou seja, **o que o software deve fazer**. Após essa fase, o próximo passo é detalhar alguns modelos da fase anterior e reagrupar em modelos de **dados**, de **comportamento**, de **interface** e de **componente**. Com isso, o "como" fazer o software deve estar complementarmente definido nesta fase de projeto.

Esses novos modelos devem seguir um **estilo arquitetural**, como o **centrado nos dados** (todos os módulos se comunicam de forma independente com o banco de dados), em **camadas** (exemplo, MVC – *Model*, *View* e *Control*), **orientado a objeto** (diagrama de componentes UML). Independente do estilo arquitetural escolhido, cada módulo definido nesta fase de projeto deve ser **coeso** (por exemplo, um módulo de cadastro de pessoas não deve gerar relatórios das pessoas cadastradas). Além disso, cada módulo deve ter pouco **acoplamento**, ou dependência entre outros módulos. Essas duas últimas características de uma arquitetura de software define a **Independência Funcional**, ou seja, é necessário buscar uma **alta coesão** e um **baixo acoplamento**.

---

[26] O Processo Unificado da Racional (RUP) foi adquirido pela IBM, para mais informações, consulte www.ibm.com/software/br/rational/

O final desta fase de projeto deve concluir um documento chamado **Especificação de Requisitos de Software** (ERS, por exemplo, seguindo na norma IEEE 830), onde todo o detalhamento de "como" o software deve ser construído, deve estar bem definido. Desta forma, um programador, ao ler esse documento, não terá dúvidas sobre como escrever o código do sistema. Um modelo para detalhar cada componente definido é apresentado a seguir:

- Nome do Componente
- Descrições
- Classes
    - Atributos – descrição
    - Métodos – descrição
        - Assinatura
        - Algoritmo: Fluxograma/Diagrama de Atividades/Pseudocódigo
- Modelo Entidade Relacionamento
- Interface entre componentes (e dentro do componente)
- Interface gráfica
- Especificações de teste
- Outros.

TESTES

O controle da qualidade de software está diretamente relacionado aos testes e à verificação dos artefatos produzidos em todo o ciclo de vida do desenvolvimento do software. Os testes se dividem em duas etapas: especificação e verificação; e execução ou validação. A especificação e a verificação ocorrem em cada fase do Requisito, Análise, Projeto e Implementação, chamados **Teste de Sistema**, **Teste de Validação**, **Teste de Integração** e **Teste Unitário**, respectivamente.

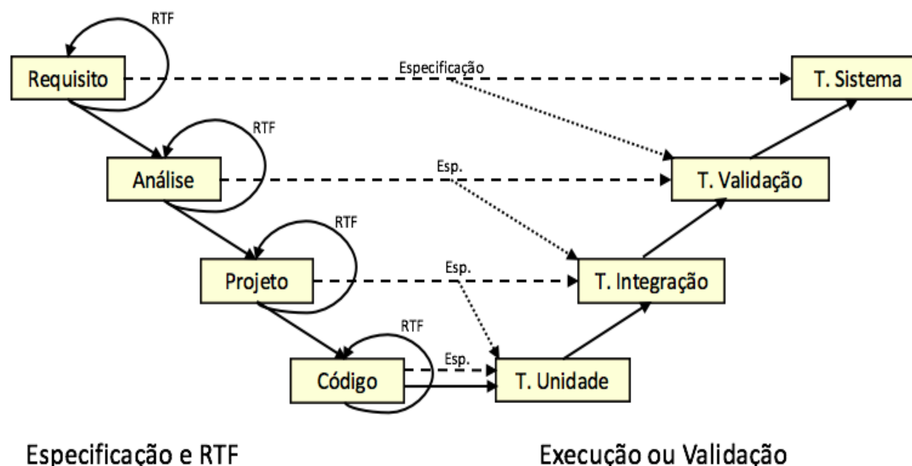


Figura 7.6. Adaptação do modelo V, focando a especificação e a execução dos testes, além da RTF.

A Figura 7.6 ilustra as várias estratégias de testes que podem existir em cada fase do ciclo de vida do desenvolvimento do software. Esta figura é uma adaptação do **Modelo V**. Este Modelo V é um detalhamento do modelo prescritivo de fluxo linear, detalhando agora os testes. No momento do levantamento de requisitos é possível fazer uma **Especificação** do **Teste de Sistema**, por exemplo, como fazer uma especificação dos testes necessários para validar uma interface gráfica. Observe que neste momento é possível fazer a especificação dos testes em protótipos de telas em papel. Nestes testes serão necessários também especificar e validar a integração do software com outros sistemas, que é uma atividade da área de Sistemas de Informação. A **Execução** ou **Validação** destes testes vão ocorrer somente no final do ciclo de vida do desenvolvimento do software. Da mesma forma, na fase de Análise, é possível finalizar a especificação do **Teste de Validação** e iniciar o **Teste de Integração**. Na fase de Projeto, quando a arquitetura será definida, será finalizado o Teste de Integração e iniciado o **Teste de Unidade**. Finalmente, na fase de Código, serão finalizadas as especificações e implementados vários Testes de Unidade, em cada módulo do sistema. Nesta fase final, é possível definir a "**complexidade ciclomática"** de cada algoritmo (métrica que define a complexidade de se testar um algoritmo e está diretamente relacionada ao número de condições lógicas).

Além das especificações e execuções dos testes, uma atividade muito importante na garantia da qualidade do software é a **Revisão Técnica Formal** (RTF), onde qualquer artefato produzido deve ser verificado usando uma *checklist*. Em média, este teste reduz em 50% os erros e RTF reduz em 75% os erros. Assim, testes e RTF devem ser atividades fundamentais em qualquer processo de desenvolvimento de software.

As atividades de testes são as que mais demandam esforço e tempo, comparado principalmente à codificação, quando o objetivo é construir um software com alta qualidade. Além disso, o custo relativo de se corrigir um erro encontrado aumenta durante o ciclo de vida do desenvolvimento do software, como ilustra a Figura 7.7.

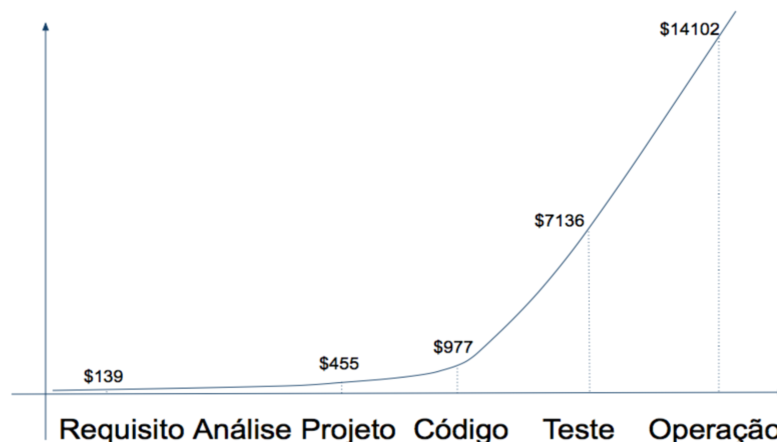


Figura 7.7. Custo relativo (US$) de corrigir um erro durante o desenvolvimento[27].

---

[27] Adaptado de Pressman, R. S.; "Engenharia de Software: Uma Abordagem Profissional".

FERRAMENTAS CASE

Uma ferramenta CASE (do inglês *Computer-Aided Software Engineering*) é um software para auxiliar no desenvolvimento dos artefatos do sistema a serem construídos. A melhor ferramenta CASE é aquela que consegue automatizar atividades do ciclo de vida do desenvolvimento do software. Alguns exemplos:

- Ferramentas para customizar processos, como RUP (do inglês, *Rational Unified Process*);
- Ferramentas para auxiliar a rastreabilidade de requisitos, ou seja, ao alterar um requisito esta ferramenta indica quais outros artefatos deverão ser alterados (ex. Rational RequisitePro da IBM);
- Ferramentas para transformar um modelo em outro modelo, como para transformar um Diagrama de Sequência em um Diagrama de Colaboração UML (ex. Rational ROSE da IBM);
- Ferramentas para transformar um modelo em código, muito comum ao gerar código usando um Diagrama de Classes ou um Diagrama de Componentes (ex. ROSE, NetBeans, Eclipse, etc.);
- Ferramentas que geram código a partir de Diagrama de Sequência, Diagrama de Atividades ou Diagrama de Estado (ex. Poseidon, Enterprise Architect, etc.);
- Finalmente, ferramentas de teste que auxiliam a automatização da validação do sistema, como o JUnit.

Assim, quando usamos ferramentas CASE, o desenvolvimento do código fica mais organizado e eficiente.

Depois dessa breve introdução à Programação Orientada a Objetos e à Engenharia de Software, é possível perceber que para construir um software com qualidade, não se deve sair escrevendo código de forma caótica, mas usando as boas práticas estudadas nesta área da ciência. Uma hora gasta num bom planejamento do software pode economizar dias gastos com codificação, testes e correção de erros.

EXERCÍCIOS

1. Complemente a classe `Fracao` com outros métodos, com o de simplificar uma fração, onde será necessário implementar o MDC e dividir ambos por ele .
2. Criar a classe `Polinomio`, tendo como atributos um vetor com os coeficientes do polinômio armazenados em um atributo do tipo vetor. Implementar vários métodos para manipular polinômios.
3. Criar a classe `Vetor`. Implementar vários métodos para manipular um vetor: inserir, remover, consultar, alterar um elemento, ordenar, condenar, somar 2 vetores (elemento a elemento, concatenar 2 vetores, etc.
4. Criar a classe `Pilha`, onde a inserção e remoção de qualquer elemento deve ocorrer em um apenas um extremo do vetor. Implementar vários métodos para manipular pilha: inserir, remover, tamanho, etc.
5. Criar a classe `Fila`, onde o primeiro elemento que entra na fila é o primeiro que sai. Implementar vários métodos para manipular fila: inserir, remover, tamanho, etc.
6. Complementar a classe `Matriz` com o método de `inverterMatriz` e `resolverSistemasEquacoes`, usando a matriz inversa.
7. Criar a classe `Imagem` e implementar vários métodos para manipular uma imagem: ler e gravar imagem do disco, transformar em imagem binária, dilatar, etc.
8. Descreva em poucas palavras processos de software.
9. Contextualize processo de software no ciclo de vida de desenvolvimento de software.
10. Relacione processo de software e engenharia de software.
11. Estabeleça uma distinção entre CMMI, processo de software, modelos prescritivos, métodos e ferramentas. Contextualize-os com os artefatos produzidos em um ciclo de vida de desenvolvimento de software.
12. Considere o sistema ***caronaUFABC*** descrito no início desse capítulo, pesquise em livros de Engenharia de Software e de UML as sintaxes dos vários diagramas citados neste capítulo e construa as várias visões de artefatos para este software.

ÍNDICE REMISSIVO



Rogério Neves

Francisco Zampirolli

Aprendendo a programar, sem complicar

Neste livro você encontrará tudo que precisa saber para começar a programar já, sem se preocupar com detalhes específicos de uma determinada linguagem, mas focando nas estruturas fundamentais que constituem um programa de computador em toda e qualquer linguagem de programação. O livro compila o conhecimento de didática no ensino de programação adquirido pelos autores em mais de 10 anos de ensino superior em ciência da computação.